فكر بايثون

كيف تفكر كعالم حاسوب

النسخة 2.0.17

فكر بايثون

كيف تفكر كعالم حاسوب

النسخة 2.0.17

ألن داوني

منشورات جرين تي

نيدهام، ماساشوستس

ترجمة طارق زيد الكيلاني

الصيغة الاصلية لهذا الكتاب هي نص أصلي من LaTeX. ترجمة نص LaTeX هذه تنتج كتاب تدريس غير معتمد على الوسيلة التي يعرض عليها، و يمكن تحويله إلى تنسيقات أخرى و طباعته.

نص LaTeX الأصلي لهذا الكتاب متوفرة من http://www.thinkpython.com.

أرسل الاعتراضات و التصحيح إن شئت الى kelany@hotmail.com.

مقدمة

التاريخ العجيب لهذا الكتاب

في كانون 1999 كنت أحضر لتدريس مدخل إلى البرمجة بلغة جافا. كنت قد درّست هذه المادة ثلاث مرات من قبل و الإحباط يتملكني. كان معدل الرسوب مرتفع جدا، وحتى من نجح من الطلاب لم يكن على المستوى المطلوب.

احدى المشاكل التي لاحظتها كانت ضخامة الكتب، و احتوائها على الكثير من المعلومات غير الضرورية،علاوة على فقرها في ارشاد الطلاب إلى كيفية البرمجة. كلها عانت من تأثير الباب المسحور: كانت تبدأ سهلة، ثم تتواصل تدريجيا، لكن عند مرحلة معينة، نحو الفصل الخامس، تنهار الارضية تحت اقدام الطلبة. كم المعلومات يتزايد بسرعة، و أضطر لإكمال الفصل محاولا لملمة الفتات.

قبل الفصل بأسبوعين، قررت كتابة المادة بنفسي. أهدافي كانت:

- **ما قل و دل.** أن يقرأ الطلاب 10 صفحات أفضل من قراءة 50.
- **الحذر عند استخدام الالفاظ.** حاولت الابتعاد عن غريب الكلام.
- **البناء التدريجي.** لتجنب الابواب المسحورة، قسمت أصعب المواد إلى سلسلة من الخطوات الصغيرة.
- **التركيز على البرمجة**، لا على لغة البرمجة. فضمنت الكتاب أقل ما يمكن من لغة جافا و تركت الباقي.

احتجت إلى عنوان، و في لحظة تجلي اخترت: **كيف تفكر كعالم حاسوب.**

كانت النسخة الاولى جلفة، الا انها أثمرت. فقد قرأ الطلاب المادة و استوعبوها بسرعة مما سمح لي باستخدام باقي زمن الفصل في شرح النقاط الأصعب و تلك الاكثر اثارة (و أكثر أهمية) تاركا الطلاب يتمرنون.

أخرجت الكتاب تحت ترخيصGNU ترخيص المنشورات المجانية، التي تسمح للمستخدمين بالنسخ و التعديل و توزيع الكتب.

ما حدث تاليا كان الجزء الأفضل. جف الكنر، مدرس ثانوي من فرجينيا، تبنى كتابي و ترجمه إلى بايثون. أرسل الي بنسخة من الترجمة، فسنحت لي الفرصة الاستثنائية لتعلم بايثون من كتابي أنا!. و كجرين تي برس، نشرت اول نسخة من بايثون في 2001.

في العام 2003 بدأت التدريس في كلية أولن، كان علي تدريس بايثون للمرة الاولى. صدمني الفرق بينها و بين جافا. قلّت معاناة الطلاب، تعلموا و عملوا أكثر على مشاريع أكثر اثارة. مختصر الحديث ..... مزيدا من المرح.

خلال التسع سنوات الماضية استمريت بتطوير الكتاب، أصحح الاخطاء، أحسن بعض الأمثلة و أضيف المواد، خصوصا التمارين.

كانت النتيجة هذا الكتاب، بعنوان أقل فخامة " فكر بايثون". بعض التغييرات كانت:

- أضفت جزءا عن تصحيح الأخطاءنهاية كل فصل. تشرح هذه الاجزاء الاساليب العامة لايجاد و تجنب البقات، و تحذيرات عن الأخطار المخفية في لغة بايثون.
- أضفت تمارين أكثر، تراوحت بين فحوصات قصيرة للتأكد من الاستيعاب إلى ما يمكن ان يرقى إلى مشاريع معتبرة.
- أضفت سلسلة من المسائل العملية – و أمثلة أطول مع تمارينها، و كذلك مناقشات. بعضها مبني على سوامبي، و هو برنامج جانبي كتبته للاستخدام خلال محاضراتي. سوامبي، الامثلة و بعض الحلول متوفرة من http://thinkpython.com.
- تعمقت في أساليب تطوير البرامج و مذاهب تصميمها الاساسية.
- أضفت تذييلات عن التصحيح، تحليل الخوارزميات، و ال " يو ام ال" مع " لمبي".

أتمنى أن تستمتع بالعمل بهذا الكتاب، و أن يساعدك في تعلم كيف تبرمج و كيف تفكر، على الاقل بدرجة واحدة اعلى، كما يفكر علماء الحاسوب.

ألن داوني

نيدهام ماساشوستس.

ألن داوني أستاذ علوم الحاسوب في كلية فرانكلن و. أولن للهندسة.

عرفان

الشكر الجزيل لجف الكنر، الذي ترجم كتابي عن جافا إلى بايثون، ما جعل هذا المشروع ينطلق. و عرفني إلى ما اصبحت لغة البرمجة المفضلة عندي.

الشكر أيضا لكرِس ميِّر الذي ساهم بعدة أقسام في "كيف تفكر كعالم حاسوب".

الشكر **لمؤسسة البرامج المجانية** FSF و إلى **التراخيص المجانية للوثائق** GNU ، التين ساعدتنا في جعل تعاوني مع جف و كرس ممكنا، و **للإبداع العام** Creative Commons على الرخصة التي استخدمها الان.

الشكر للمحررين في لولو Lulu الذين عملوا على "كيف تفكر كعالم حاسوب".

الشكر لكل الطلبة الذين عملوا بالنسخ الأبكر من هذا الكتاب و كل المساهمين (في القائمة أدناه) الذين أرسلوا التصحيحات و الاقتراحات.

قائمة المساهمين

أكثر من مئة من ثاقبي النظر و من القراء المفكرين أرسلوا مقترحاتهم و تصحيحاتهم خلال السنوات الماضية. مساهماتهم و حماسهم لهذا المشروع، كانت دعما كبيرا.

إن كان لديك اقتراح أو تصحيح، فالرجاء أرساله إلى البريد `feedback@thinkpython.com` ، و إن قمت بتغيير ما بناءا على ما أرسلته سأضيف اسمك إلى قائمة المساهمين (إلا إن أردت عدم الاضافة).

إن ضمَّنت جزءا على الاقل من العبارة التي يظهر فيها الخطأ ستسهل عليّ البحث. رقم الصفحة و القسم يفيد أيضا لكن ليس بسهولة البحث عن عبارة محددة. شكرا!

لويد هف ألن أرسل تصحيحا على القسم 8.4.

إيفون بويان أرسل تصحيحا لخطأ دلالي في الفصل الخامس.

فرد بريمر أرسل تصحيحا في القسم 2.1.

يونا كوهن كتب نص بيرل لتحويل النص الأصلي من لاتكس لهذا الكتاب إلى HTMLجميل.

ميشيل كنلن أرسل تصحيحا قواعديا على الفصل الثاني و تحسينات على الأسلوب في الفصل الاول، و هو من استهل النقاشات حول النواحي الفنية للمفسِّرات .

بنوا جيرار أرسل تصحيحا لخطأ مضحك في القسم 5.6.

كورتني جليسون و **كاثرن سمث** كتبا horsebet.py و الذي استخدم كدراسة حالة في نسخ سابقة لهذا الكتاب. برنامجهما متوفر الان على موقع الويب.

لي هر أرسل من التصحيحات مما لا يتسع لذكرها المقام، و لذا يجب أن يدرج كأحد المحررين الرئيسين لهذا المخطوط.

جيمس كيلن طالب يستخدم هذا الكتاب و قد أرسل العديد من التصحيحات.

ديفد كيرشو أصلح اقتران catTwice كان مكسورا في القسم 3.10

إدي لام أرسل العديد من التصحيحات على الأقسام 1،2 و 3 . و أصلح أيضا Makefile بحيث تنشئ مؤشرا عندما تشغل لأول مرة و ساعدنا في تجهيز طريقة تسمية النسخ.

مان-يونغ لي أرسل تصحيحا للمثال في القسم 2.4

ديفد مايو أوضح لنا بأن الكلمة unconsciously في الفصل الأول يجب أن تستبدل بـ.subconsciously

كرس مكلون أرسل العديد من التصحيحات على الاقسام 3.9 و 3.10

ماثيو جي مولتر كان مساهما دائما و قد أرسل العديد من التصحيحات و الاقتراحات لهذا الكتاب.

سيمون ديكون مانفورد أبلغنا عن تعريف إقتران نسيناه و أخطاء مطبعية عديدة في الفصل الثالث. و قد عثر أيضا على أخطاء في الاقتران المزيدي في الفصل 13

جون اوزتس صحح تعريف القيمة المرتجعة في الفصل الثالث

كفن باركس أرسل تعليقات قيمة و اقتراحات على كيفية رفع مستوى توزيع الكتاب.

ديفد بول أرسل لنا عن خطأ مطبعي في قسم المعاني في الفصل الاول، و كذلك كلمات لطيفة كتشجيع.

مشيل شمت أرسل تصحيح على فصل الملفات و الاستثناءات

روبن شو عين لنا مكان خطأ في القسم 13.1 حيث استخدم printTime في مثال بدون تعريفه.

بول سليخ عثر على خطأ في الفصل 7 و بقة في نص بيرل لـ يونا كوهن الذي يولد HTML من لاتكس

كريغ تي سيندل يختبر المخطوط في دورة في جامعة درو. لقد ساهم بالعديد من الاقتراحات القيمة و التصحيحات.

إيَن تومس و طلبته يستخدمون المخطوط. هم أول من يختبر الفصول في النصف الاخير من الكتاب و قد قدموا العديد من التصحيحات و الاقتراحات.

كيث فرهيدن أرسل تصحيحا على الفصل الثالث.

بيتر وستنلي أعلمنا عن خطأ قائم منذ مدة في النص اللاتيني المستخدم في الفصل 3.

كرس ربل قام بالعديد من التصحيحات على النص البرمجي في الفل مدخلات و مخرجات الملفات و الاستثناءات.

موشي زدكة مساهمته قيمة لهذا المشروع. فبالإضافة إلى كتابة المسودة الاولى لفصل القواميس، قدم أرشادات قيمة منذ المراحل الاولى لهذا الكتاب.

كرستوف زورِشكِة أرسل العديد من التصحيحات و اقتراحات في التعليم و شرح الفرق بين gleich و.selble

جيمس ميَّر أرسل لنا دلوا ممتلئا من الأخطاءاللفظية و المطبعية من ضمنها خطأين في قائمة العرفانات نفسها.

هايدن مَكَفي التقط ما قد يصبح تناقض مشوش بين مثالين.

هايدن أرنل هو من الفريق العالمي من المترجمين و يعمل على النسخة الاسبانية من المخطوط. و قد عثر على عدة أخطاء في النسخة الانجليزية

توحيد الحق و **لكس بيرزني** أنشا الأشكال التوضيحية في الفصل الاول و حسّنا الكثير من الاشكال الاخرى

د. ميشيل الزيتا التقط خطأ في الفصل 8 و أرسل بضعة اقتراحات مميزة حول التعليم و اقتراحات حول فيبوناشي وOld Maid

اندي متشل التقط خطأ مطبعيا في الفصل الاول و مثال مكسور في الفصل الثاني.

كالن هارفي اقترح توضيحا في الفصل 7 و التقط بضعة أخطاء مطبعية.

كرستفر بي سمث التقط عدة أخطاء مطبعية و ساعد في تحديث الكتاب إلى بايثون 2.2

ديفد هطشنز التقط خطأ مطبعيا في المقدمة.

غريغور لنغل يعلم بايثون في معهد عال في فينا، النمسا. و هو يعمل على الترجمة الألمانية للكتاب، و قد التقط زوجان من الأخطاءالسيئة في الفصل 5.

جولي بيترز التقطت خطأ مطبعي في المقدمة

فلورين أبرينا أرسلت تحسينا لـ makeTime و تصحيحا لـ .printTime و خطأ مطبعي جميل.

دي جي ويبر اقترح توضيحا في الفصل 3.

كن وجد حفنة من الأخطاءفي الفصول 8،9 و 11.

إيفوويفر التقط خطأ مطبعي في الفصل 5 و اقترح توضيحا في الفصل 3

كرتس يانكو اقترح توضيحا في الفصل 2

بن لوغن أرسل لنا عن العديد من الأخطاءالمطبعية و مشاكل ترجمة هذا الكتاب إلى .HTML

جيسن ارمسترنغ رأي الكلمة المفقودة في الفصل 2

لويس كورديه لاحظ موضعا في الكتاب حيث النص البرمجي لم يطابق المكتوب.

برين كين اقترح عدة توضيحات في الفصل 2 و 3.

رب بلاك أرسل سلة من التصحيحات من ضمنها التغييرات في بايثون 2.2

جان-فيلب ريه من Ecole Centrale Paris أرسل عدد من المعلومات الصغيرة من ضمنها بعض التحديثات على بايثون 2.2 و تحسينات معتبرة.

جيسن مادر من جامعة جورج واشنطن قام بعدة إقتراحات مفيدة و تصحيحات.

يان قندفتة-برون ذكرنا بأن "a error" هي.error

ابل ديفد و **الكسس دينو** ذكرانا بأن جمع matrix هو matrices و ليس matrixes ظل هذا الخطأ في الكتاب لسنوات، لكن قارئين بنفس الحروف في أول أسمائهما نبهانا له في نفس اليوم، غريب.

شارلز تاير شجعنا لكي نتخلص من الفاصلة المنقوطة التي وضعناها في نهاية بعض العبارات و لأن ننظف طريقتنا في استخدام كلمات argument و parameter.

روجر سبربرغ أبلغنا عن قطعة منطقية غريبة الاطوار في الفصل 3.

سام بُل أبلغنا عن فقرة مشوشة في الفصل 2.

اندرو شيونغ تجليتان لـ الاستخدام قبل التعريف.

سي كوري كيبل لاحظ كلمة مفقودة في الفرضية الثالثة في تصيد الأخطاءو الأخطاءالمطبعية في الفصل 4

ألساندرة ساعدتنا في توضيب بعض الالتباس في السلحفاة

وم شمبين وجد brain-o في مثال القاموس

دغلس رايت أبلغ عن مشكلة في القسمة الأرضية فيarc

جيرد سبنذُر وجد بعض المطروحات في نهاية عبارة.

لِن بيهنغ أرسل عددا من الاقتراحات التي تساعد.

راي هاختفدت أرسل عن خطأين و عن ليس-خطأ-بمعنى الكلمة.

تورستن هوبش أبلغ عن تضارب في سومبي

إنغة بتوهوف صححت مثال في الفصل 14.

أرنِة بابنهوزنهيد أرسل العديد من التصحيحات المفيدة.

مارك إي كاسيدة ماهر في ملاحظة الكلمات المكررة.

سكوت تايلر ملأ "كان هذا مفقودا" ثم أرسل كومة من التصحيحات.

جوردُن شبرد أرسل العديد من التصحيحات، كلها في رسائل إلكترونية منفصلة.

اندرو تيرنر لاحظ خطأ في الفصل 8.

ادم هُبرت أصلح مشكلة في القسمة الارضية فيarc

دارل هموند و **سارة زِمَرمن** لاحظتا أنني قدمت math.py مبكرا. و زم لاحظت خطأ مطبعي.

جورج ساس وجد بقة في قسم تصيد البق (علاج الاخطاء)

برَيَن بنغهام اقترح التمرين 1.10

ليئة إنغلبرت-فنتون نتبهت إلى أنني استخدم tuple كاسم متغير، على خلاف نصيحتي أنا. و عثرت على كمية من الأخطاءالمطبعية و "الاستخدام قبل التعريف"

جو فونكة وجد خطأ مطبعي.

تشاو-تشاو تشن وجد تناقضا في مثال فيبوناشي

جف بين يعرف الفرق بين space و spam

لوبُس بنتس أرسل خطأ مطبعي

غريغ لند و **أبيغيل هيتهُف** اقترحا التمرين 14.4

ماكس هيلبرن أرسل العديد من التصحيحات و الاقتراحات. ماكس هو أحد مؤلفي " التجريد المتماسك" العظيم، الذي قد تود قراءته بعد انتهائك من هذا الكتاب.

شوتيبات بورنهافالي وجد خطأ في رسالة خطأ

ستانِسلوف أنتول أرسل اقتراحات مفيدة للغاية.

إريك بشمن أرسل العديد من التصحيحات في الفصول 11-4.

مِغوِل أزِفيدو وجد بعض الأخطاءالمطبعية.

جيانهو ليو أرسل قائمة طويلة بالتصحيحات.

نِك كنغ وجد كلمة مفقودة.

مارتن زوثر أرسل قائمة طويلة من المقترحات.

ادم زِمَرمن وجد تناقضا في تجليتي لتجلية و غيرها العديد من الاخطاء.

راتناكار تيواري اقترح تذييلا يشرح تحلل المثلثات.

أنوراغ غول اقترح حلا اخر لـ is_abecedarian و أرسل بعض التصحيحات الاضافية. و هو يعلم كيف يهجئ Jane Austen.

كيلي كراتزر نتهت إلى خطأ مطبعي.

مارك غرِفِثس نبه إلى مثال محير في الفصل 3

رويدن أونجي وجد خطأ في طريقة نيوتن خاصتي.

باتريك ولوويتش ساعدني في مشكلة في نسخة HTML.

مارك شونفسكي أخبرني عن كلمة مفتاحية جديدة في بايثون.

رَسِل كولِمن ساعدني في حساب المثلثات.

ويه هوانغ عثر على عدة أخطاء مطبعية.

كارن بربر لاحظت أقدم ال الأخطاءالمطبعية في الكتاب.

نام نغوين وجد خطأ مطبعي و نبه إلى أنني أستخدم نمط التزيين دون الاشارة إلى اسمه.

ستِفَني مورِن أرسلت العديد من التصحيحات و الاقتراحات.

بول ستوب صحح خطأ مطبعي في uses_only

إريك برونر نبه إلى تشويش في المناقشة حول ترتيب العمليات.

ألِخاندرو خِزِرلس وضع قاعدة جديدة لنوعية و عدد الاقتراحات التي يرسلها. نحن ممتنون للغاية!

جري تومس يعلم يساره من يمينه.

جيوفاني إسكُبار سوسة أرسل قائمة طويلة من التصحيحات و الاقتراحات.

ألكس إتين أصلح إحدى الـ URL.

كوانغ هي وجد خطأ مطبعي.

دانيل نلسن صحح خطأ عن ترتيب العمليات.

وِل مكجنس نبه إلى إختلاف تعريف polyline في مكانين.

سوارُب ساهو عثر على فاصلة منقوطة مفقودة.

فرانك هيكر نبه إلى مثال لم يحدد جيدا و إلى بعض الروابط المقطوعة.

أنيمش B ساعدني في تنظيف مثال مشوش.

مارتن كاسبرسن وجد خطأين ختاميين.

غريعور ألم أرسل عدة تصحيحات و اقتراحات.

دمتريوس تسيريغس اقترح علي توضيح تمرين.

كارلوس تافور أرسل صفحة من التصحيحات و الاقتراحات.

مارتن نوردسلتن وجد بقة في حل أحد التمارين.

لارس أو دي كرستيَنسن وجد مرجعا مقطوعا.

فكتور سيمون وجد خطأ مطبعي.

سفِن هوكستر نبه إلى أن متغيرا يسمة input له ظل اقتران جاهز.

فيت لي وجد خطأ مطبعي.

ستِفِن غريغوري نبه إلى مشكل مع cmp في بايثون 3.

ماثيو شولتز أبلغني عن رابطة مقطوعة.

لوكش كومار مكاني أبلغني عن بعض الروابط المقطوعة و عن بعض التغييرات في رسائل الخطأ.

إشوار بهات صحح لي عبارتي عن نظرية فِرمات الاخيرة.

براين مكغي اقترح توضيحا.

أندريا زانيللا ترجم هذا الكتاب إلى الإيطالية.، و أرسل إلي عدد من التصحيحات في طريقه.

المحتويات

الفصل الاول

طريق البرنامج

هدف هذا الكتاب تعليمك كيف تفكر كعالم حاسوب، تفكير يجمع بين بعضٍ من افضل خصائص الرياضيات و الهندسة و العلوم الطبيعية. فمثل علماء الرياضيات، يستخدم عالم الحاسوب لغة رسمية لايصال الافكار (خصوصا الحسابية). و كما المهندسين، يصممون الأشياء و يقومون بتجميع المكونات لتصبح نظما ثم يقيّمون افضل الخيارات من بين العديد من البدائل. و كالعلماء، يراقبون سلوك الأنظمة المعقدة، و يضعون الفرضيات، و التنبؤات.

المهارة الأهم لعالم الحاسوب هي **حل المشكلات**، حل المشكلات يعني تفسيرها بصيغة رسمية،ثم التفكير الخلاق بالحلول، ثم التعبير عن الحل تعبيرا واضحا و دقيقا. الحق أنَّ عملية تعلم البرمجة هي فرصة تُقتنَص للتمرن على المهارات العديدة لحل المشكلات، و لهذا سمي هذا الفصل " طريق البرنامج".

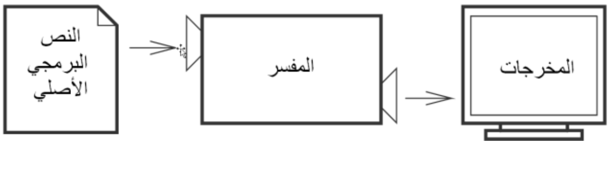
فمن ناحية ستتعلم البرمجة، و هي مهارة مفيدة لذاتها، و من ناحية اخرى ستستخدم البرمجة كوسيلة لهدف ابعد، ستتضح لك معالمه كلما تقدمنا معا.

1.1 لغة البرمجة بايثون

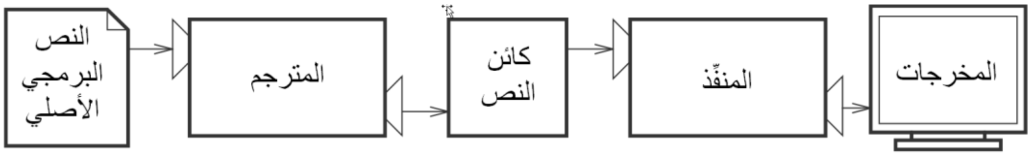
لغة البرمجة التي ستتعلمها هنا هي بايثون. و بايثون مثال **للغات البرمجة عالية المستوى**، قد تكون قد سمعت عن لغات برمجة عالية المستوى مثل سي، سي بلس بلس، بيرل و جافا.

يوجد، ايضا ما يسمى **بلغة منخفضة المستوى**، تدعى احيانا لغة الالة او لغة التجميع. الحواسيب تشغل البرامج المكتوبة باللغات منخفضة المستوى فقط، لذلك فان البرامج المكتوبة بلغة عالية المستوى لن تعمل قبل معالجتها. هذه المعالجة تستنفذ بعض الوقت، لكنه ليس سوى سيئة صغيرة للغات عالية المستوى.

حسنات اللغات عالية المستوى لا تحصى، **أولاها:** سهولة كتابة البرامج. فكتابة البرامج باللغات عالية المستوى تتطلب وقتا اقل، و البرامج المكتوبة تكون اقصر و اسهل للقراءة، و احتمال خلوها من الأخطاء اكبر. **ثانيها:** قابليتها للنقل، اي ان البرنامج المكتوب بلغةٍ عالية المستوى سيعمل على انواع مختلفة من الحواسيب بتعديل بسيط او حتى بدون اي تعديل. اللغات منخفضة المستوى تعمل على نوع واحد من الحواسيب، و يجب كتابتها مرة اخرى لتعمل على نوع اخر.



الشكل 1.1: المفسّر يعالج البرنامج قليلا كل مرة، فيقرأ سطور البرنامج و ينفذ العمليات



الشكل 1.2: المترجم يترجم النص الاصلي إلى إلى تشفير شيئي، ثم تنفذه مكونات الحاسوب المادية

لأجل هذه المزايا، تكتب جميع البرامج تقريبا باللغات عالية المستوى، بينما تستخدم اللغات منخفضة المستوى في تطبيقات متخصصة.

هنالك نوعان من البرامج التي تعالج اللغات عالية المستوى لتصبح لغة آلة: **المفسّرات** و **المترجمات**. المفسّر يقرأ سطور برنامج مكتوب بلغة عالية و ينفذها، اي انه ينفذ ما يقول البرنامج معالجا اوامره واحدا تلو الاخر. فيقرا السطور و يقوم بالعمليات الحاسوبية. الشكل 1.1 يبين هيكل المفسّر.

اما المترجم فيقرا البرنامج و يترجمه كاملا قبل تشغيله. في هذا السياق يسمى البرنامج المكتوب بلغة عالية المستوى **بالنص الأصلي** و البرنامج المترجم **بالمشفر الشيئي** أو **البرنامج القابل للتنفيذ**. بمجرد ترجمة البرنامج سيمكن تشغيله مرارا دون الحاجة إلى الترجمة. الشكل 1.2 يبين هيكل المترجم.

يعتبر بايثون مفسّرا، البرامج المكتوبة تنفذ عن طريق مفسِّر. هنالك طريقتان لاستخدام المفسّر: **الوضع التفاعلي** و **وضع كتابة النص.**

في الوضع التفاعلي ستكتب برنامج بايثون، و المفسّر سينفذه لك و يظهر النتيجة:

```
>>> 1 + 1
2
```

اشارات <<< (المحث) تعني ان المفسّر جاهز، فان طبعت 1+1 فسيرد المفسّر بـ 2.

في المقابل، يمكنك تخزين النص الأصلي في ملف ثم استخدام المفسّر لتنفيذ محتواه. ما اجتُمِع عليه هو ان ملفات بايثون تنتهي بـ (`.py`). و لكي يُنفذ النص عليك ابلاغ بايثون باسم الملف. فلو كان لديك نصا مخزنا تحت اسم `dinsdale.py` و كنت تعمل على **UNIX** فستطبع `python dinsdale.py`. تختلف هذه التفاصيل من بيئة تشغيل إلى اخرى، بإمكانك الحصول على معلومات بهذا الخصوص من موقع `http://python.org`.

العمل في الوضع التفاعلي مفيد لفحص القطع البرمجية الصغيرة، لأنك تستطيع تنفيذها مباشرة. لكن لكل ما هو اكبر من بضعة سطور سيكون عليك تخزين النص بحيث يمكنك تعديله و تنفيذه مستقبلا.

1.2 ما هو البرنامج؟

البرنامج هو مجموعة تعليمات مرتبة، تحدد طريقة تنفيذ عملية **حوسبة**. الحوسبة قد تكون شيئا حسابيا، كحل منظومة من المعادلات او ايجاد جذور كثيرات الحدود، إلا انها قد تكون حوسبة رمزية، كالبحث و استبدال النصوص في وثيقة او (و يا للعجب) ترجمة برنامج.

تختلف تفاصيل التعليمات باختلاف اللغات، إلا ان بضعة تعليمات اساسية ستتبدى تقريبا في كل لغة:

المُدخَلات: الحصول على المعلومات من لوحة المفاتيح او الملفات او اي اداة اخرى.

المُخرَجات: اظهار المعلومات على الشاشة او ارسالها إلى ملف او اي اداة اخرى.

الحساب: القيام بالعمليات الحسابية الاساسية كالجمع و الضرب.

التنفيذ المشروط: تنفيذ عملية ما إن تحققت الشروط.

التكرار: القيام بعمل ما مِرارا، و عادة ما يكون مع بعض التغيير.

صدّقت ام لا، هذا تقريبا كل ما ينطوي عليه البرنامج. كل برنامج استخدمته، بغض النظر عن مدى تعقيده، بني على تعليمات كهذه، لذا يمكنك التفكير في البرمجة على انها تقسيم لمهمة ضخمة معقدة إلى مهمات اصغر فاصغر، حتى تصبح بسيطة إلى الحد الذي يمكّنِنا من تنفيذها بهذه التعليمات.

قد لا يزال هذا غامضا، الا اننا سنعود إلى هذه النقطة عند الحديث عن الخوارزميات.

1.3 تصحيح الاخطاء، ما المقصود بـ Debugging ؟

البرمجة عرضة للخطأ، و لأمر غريب تدعى اخطاء البرمجة بق (كتلك الحشرة) و عملية صيد الأخطاء تدعى Debugging (أو تدعى قمل، و تصيدها تفلية).

هنالك ثلاثة انواع من الأخطاء التي قد تحدث خلال البرمجة: **اخطاء نحوية**، **اخطاء عند التشغيل** و **الأخطاء الدِلالية**. من المفيد التفريق بينها لتسريع عملية صيد الأخطاء.

1.3.1 الأخطاءالنحوية Syntax Errors

بإمكان بايثون تنفيذ البرامج فقط إن كانت خالية من الأخطاء النحوية، و إلّا فسيطبع مفسِّر بايثون رسالة وجود خطأ يتعلق بالأخطاء النحوية. الأخطاء النحوية تشير إلى بناء البرنامج و القوانين التي تحكم نحوه، فالأقواس مثلا لا تقبل إلّا اذا كانت ازواجا، فـ (1+2) قانونية، اما (8 فهي خطأ نحوي.

في العربية، يتسامح القراء مع معظم الأخطاء النحوية، هل لاحظت أنك تقرأ رسائل على مواقع التواصل الإجتماعي مليئة بالأخطاء النحوية، لكنك تتغاضى و لا ترسل Error message.

في المقابل، فإن بايثون ليس متسامحا بالمرّة، فبوجود خطأ واحد في البرنامج سيطبع رسالة وجود خطأ و يتوقف عن التنفيذ، و لن يكون بامكانك تشغيل البرنامج.

خلال الاسابيع الاولى في حياتك البرمجية سيأخذ تصيد و تصحيح الأخطاء الكثير من وقتك ، لكن مع اكتسابك للخبرة، ستقوم بأخطاء اقل و ستعالجها بشكل اسرع.

1.3.2 الأخطاء التي تظهر عند التشغيل Runtime Errors

النوع الثاني من الأخطاء سمي كذلك لأن هذه الأخطاء لا تظهر خلال التصميم، بل بعد تشغيل البرنامج. هذه الأخطاء تسمى ايضا **استثناءات** لأنها تؤشر بحدوث امر استثنائي (سيء).

1.3.3 الأخطاء الدلالية (التأويل) Semantic Errors

ان كان بالبرنامج **خطأ دلالي** فإن البرنامج سيعمل بنجاح - من ناحية عدم اصدار رسائل بوقوع خطأ ما- إلّا أنه لن يقوم بالعمل المطلوب، بل سيقوم بعمل آخر. أو للدقة: سيقوم بما طلب منه القيام به.

ستكون المشكلة ان البرنامج الذي قمت بكتابته ليس ذلك الذي اردت كتابته. فمعنى البرنامج (دلاليته) خطأ. التعرف على الأخطاءالدلالية كالتعرف على الخدع لأنه يتطلب منك العمل بشكل عكسي، سيكون عليك تأمل مخرجات البرنامج ثم محاولة معرفة ما يقوم به.

1.3.4 تصيد الأخطاء تجريبيا

من أثمن المهارات التي ستحصل عليها هي تصيد الأخطاء و تصحيحها، و مع أنها محبطة، إلا ان هذه المهارة هي الجزء الاكثر تحديا و اثارة و اثراءا للثقافة الذاتية.

فمن ناحية، يشبه تصيد الأخطاء عمل التحري، هنالك أدلة بين يديك و عليك تخمين الأحداث و العمليات التي ادت لوقوع الخطأ الذي تراه.

تصيد الأخطاء كالعلوم التجريبية، فبمجرد ورود فكرة في رأسك عما أوصل إلى هذا الخطأ، ستبدأ بتعديل البرنامج و تجربته مرة اخرى، فإن كانت افتراضاتك صحيحة ستتمكن من التنبؤ بنتيجة التعديل الاخير ثم تقوم بتعديل آخر يقربك من برنامجك المقصود. و ان كانت افتراضاتك خاطئة، سيكون عليك الإتيان بفرضية اخرى. فكما بيَّن شيرلوك هولمز"إن أزلت من المستحيلات، فما يتبقى سيكون الحقيقة مهما بدت لا تصدق" (أكونان دويل، علامة الاربعة).

البعض يرى البرمجة و تصيّد الأخطاء شيئا واحدا، لكون البرمجة تصيدا متواصلا للأخطاء إلى تنتهى إلى برنامج يقوم بما تريده ان يقوم به. الفكرة هي ان عليك البدء ببرنامجٍ يقوم بشيء ما، ثم تبدأ بإضافة التعديلات الصغيرة و تصيُّد اخطائها و تصحيحها بينما تسير في طريقك نحو البرنامج المكتمل، فيكون امامك برنامجا خاليا من الأخطاء طوال عملية البناء.

على سبيل المثال، Linux نظام تشغيل يحتوي على الالاف من سطور البرمجة، الا انه بدأ ببرنامج صغير بناه لينوس تورفالدز لكي يسكشف شريحة انتل 80386 . و حسب لاري جرينفيلد "احدى مشاريع لينوس الاولى كانت برنامجا بامكانه التبديل بين طباعة AAAA و BBBB. تطور هذا البرنامج لاحقا إلى Linux" (من The Linux Users' Guide Beta Version 1).

في الفصول الاخيرة تنتظرك اقتراحات اكثر حول تصيد الأخطاء، و كذلك حول الممارسات الصحيحة في البرمجة.

1.4 اللغات الرسمية و اللغات الطبيعية:

اللغات الطبيعية هي ما يتحدث به الناس، كالعربية و الانجليزية، لم يصمم الناس هذه اللغات (رغم محاولاتهم وضع القواعد لها)، هذه اللغات تطورت طبيعيا.

اللغات الرسمية: لغات صممها الناس لتُطبَّق في أجواء معينة. الإشارات الحسابية التي يستخدمها الرياضي هي لغة رسمية تُستخدم في تحديد العلاقة بين الأرقام و الرموز. الكيميائيون يستخدمون لغة رسمية للتعبير عن تركيب الجزيء و الأهم:

لغات البرمجة هي لغات رسمية صممت للتعبير عن عمليات الحوسبة

للنحو في اللغات الرسمية قواعد صارمة، فـ 6 = 3 + 3 صيغة صحيحة في حين 3$6 = +3 صياغة خاطئة، كذلك H_2O صيغة صحيحة لمعادلة كيميائية لكن Zz_2 ليست كذلك.

قواعد النحو تأتي بنكهتين، تتعلقان **برموز اللغة** token و بنائها. رموز اللغة هي عناصر اللغة الاساسية، كالكلمات و الأرقام و العناصر الكيميائية. فالمشكلة في 3$6 = +3 هي أن $ ليست رمزا قانونيا في الرياضيات (على الأقل على حد علمي) كذلك Zz_2 ليست قانونية لعدم وجود عنصر يرمز له Zz.

النكهة الثانية من نحو اللغة تتعلق ببنائها، أي كيف ترتب رموز اللغة. فالعبارة 3 = + 3 غير صحيحة رغم كون + و = رموزا رياضية قانونية، إلا اننا لا نستطيع وضعها إلى جانب بعضها، هذا ينطبق على المعادلات الكيميائية ايضا، فالارقام المنخفضة تأتي بعد رموز العناصر لا قبلها.

تمرين 1.1 : اكتب جملة عربية صحيحة القواعد لكن بكلمات غير صحيحة، ثم اكتب اخرى بكلمات صحيحة و بنحو غير قانوني.

إن كنت تقرأ جملة عربية أو عبارة ذات صياغة رسمية، سيكون عليك تبيُّن نحو الجملة (علما بأنك تفعل هذا مع اللغات الطبيعية دون ان تنتبه) هذا ما يسمى في العربية "الإعراب و الصرف" في الانجليزية Parsing سنتحدث من الان عن الانجليزية فقط!!

فمثلا عند سماعك "The penny dropped" ستفهم بأن "The penny" هو الموضوع (subject) و أن "dropped" هو المتممة للجملة (predicate). بمجرد اعرابك للجملة سيكون بامكانك فهمها، أو معرفة "دلالتها" its semantic ، هذا الفهم مشروط بكونك تعلم ما هو الـ Penny و معنى السقوط.

تتشابه اللغات الرسمية و الطبيعية في نواح عدة، الا انها تختلف في بعض الامور:

الغموض: اللغات الطبيعية يلفُّها الغموض، يتحايل الناس على هذا بالادلة التي تفهم من السياق. اللغات الرسمية صممت لتكون واضحة تماما، مما يعني أن لكل عبارة معنىً واحد فقط، بغض النظر عن السياق.

الوفرة: تمتلك اللغات الطبيعية الكثير من الكلمات لكي تعوض غموضها و لتجنب الفهم الخاطئ، و نتيجة لهذا غالبا ما تكون كثيرة الاطناب. اللغات الرسمية أفقر و أوجز.

المعنى الحرفي: اللغات الطبيعية مليئة بالتكنية، فعندما نقول "القشة التي قصمت ظهر البعير"، قد لا يكون هناك بعير و لا قشة و لا شيء قد كسر اساسا. اللغات الرسمية تعني بالضبط ما تقوله.

الناس الذين تربوا على استعمال اللغات الطبيعية (كل الناس) يجدون صعوبة في التحول إلى اللغات الرسمية، أحيانا يكون الفرق بينها كذلك الذي بين الشعر و الخطابة، بل أكبر:

الشعر: تُستعمل الألفاظ لصوتها و لمعناها، القصيدة ككل تُعمل تأثيرا و ردة فعل عاطفية، و غالبا ما يكون الغموض في المعاني الشعرية مقصودا.

الخطابة: المعنى الصريح للعبارات هو المهم هنا، طريقة نحو الجمل تساهم في المعنى الاجمالي، تحليل الخطب اسهل من تحليل القصائد إلّا انها تظل تحتوي على الغموض.

البرنامج: معنى برنامج الحاسوب واضح و حرفي، يمكن فهمه كليا عن طريق تحليل رموزه و بنائه.

اليك الان بعض الاقتراحات لقراءة البرامج (و كذلك اللغات الرسمية الاخرى). أولا، تذكر ان اللغات الرسمية مكثفة على عكس الطبيعية، لذا فقراءتها ستأخذ وقتا اطول ، ثم تذكر بأن بنيان البرنامج في غاية الاهمية، لذلك فبدلا من قراءة البرنامج من الاعلى إلى الاسفل و من اليسار إلى اليمين، تعلم تحليل هيكل البرنامج في رأسك و تعريف رموز البرنامج و تفسير بناءه. اخيرا، انتبه للتفاصيل، فالأخطاء الإملائية التي غالبا ما نتساهل معها في اللغات الطبيعية، لها تبعات أعظم في

اللغات الرسمية.

1.5 البرنامج الاول

تقليديا، اول برنامج تكتبه بلغة برمجة جديدة يسمى "Hello, World!"، لأن كل ما سيفعله هو اظهار هذه الجملة "Hello, World!" على الشاشة. في بايثون يكتب هذا البرنامج كالتالي:

```
print 'Hello, World!'
```

هذا مثال على عبارة print، رغم ان print تعني "اطبع" فهذا البرنامج لا يطبع اي شيء على الورق، فقط يظهر قيمة على الشاشة. القيمة في هذه الحالة هي الكلمات:

```
Hello, World!
```

علامات الاقتباس (') تحدد بداية و نهاية النص المطلوب عرضه، و لن تظهر عند عرض النتيجة.

في بايثون3، تختلف الصيغة قليلا:

```
Print('Hello, World!')
```

الاقواس هنا تعني أن `print` هي اقتران، سنناقش الاقترانات لاحقا في الفصل الثالث.

سأستخدم `print` كعبارة حتى نهاية الكتاب، فان كانت نسختك بايثون3 فعليك ترجمتها. ما عدا ذلك فالفروق بين النسختين قليلة فلا تقلق بسببها.

1.6 تصيد الاخطاء

الافضل قراءة هذا الكتاب مقابل حاسوبك لكي تتمكن من تطبيق الامثلة مباشرة، بإمكانك تجربة معظم الامثلة في الوضع التفاعلي، إلا انه اذا كتبت البرنامج في الوضع النصي ستتمكن من تجريب التعديلات عليه.

عندما تتدرب على أوامر جديدة عليك ان تحاول تعمُّد الخطأ، فمثلا ما الذي سيحدث في برنامج "Hello, World!" ان نسيت و ضع احدى علامتي الاقتباس؟ أو كلاهما؟ ما الذي سيحدث لو أخطأت تهجئة كلمة `print`؟

هذا النوع من التجريب يساعدك على تذكر ما تقرأ هنا، و سيساعدك عند تصيد الاخطاء، فلكي تتذكر دائما ما الذي تقصده

الرسالة التي تفيد بوجود خطأ ما، من الافضل القيام بذلك الخطأ الان و عن قصد بدلا من انتظار حدوثه عرضيا.

أحيانا تثير البرمجة بعض المشاعر ، خصوصا تصيد الاخطاء، فقد تحس بالإحباط عندما يواجهك خطأ برمجي عصي على الحل، و قد تشعر حتى بالغضب.

هنالك دلائل على ان البشر قد يتعاملون مع الحواسيب كما لو كانت بشرا مثلهم، فان كانت تعمل جيدا فهي جزء من الفريق وإلّا فهي فظة، و منهم فردّات فعلهم ستكون كتلك نحو الفظين من الناس (ريفز و نس، معادلة الميديا: كيف يعامل الناس الحواسيب، التلفزيونات و الميديا الجديدة كالناس و الأمكنة).

قد يساعدك ان تتحضر لردات الفعل هذه في التعامل معها. احدى الطرق هي اعتبار الحاسوب احد موظفيك، له نقاط قوة محددة، كالسرعة و الدقة، و له كذلك نقاط ضعف معينة كعدم الاحساس مع الاخرين و عدم بعد النظر.

مهمتك هي ان تكون المدير الجيد، استفد إذن من مهارات و دقة هذا الموظف ثم تقبّل فظاظته و عدم رؤيته للصورة الكاملة، ثم جِد السُبل لاستخدام عواطفك من غضب و خجل في حل المشاكل، دون ان تسمح لردات فعلك من أن تنتقص من كفاءتك في العمل.

الاحباط من سمات التدرب على مهارة صيد الاخطاء، الا ان قيمة امتلاك هذه المهارة تتخطى تصحيح اخطاء البرامج التي تكتبها إلى كثير من فعالياتك اليومية. في نهاية كل فصل من فصول هذا الكتاب، هناك قسم عن تصيد الاخطاء، كهذا، و به ستجد كيف افكر عندما اتصيد الاخطاء، اتمنى ان يكون ذا فائدة لك.

1.7 معاني:

حل المشاكل problem solving: عملية متكاملة، تشمل صياغة المشكلة، ايجاد حل لها، ثم التعبير عن هذا الحل.

لغات عالية المستوى high-level language: لغات برمجة كبايثون مصممة لكي تكون سهلة القراءة و الكتابة.

لغات منخفضة المستوى low-level languag: لغات برمجة مصممة لكي ينفذها الحاسوب بسهولة، تسمى أيضا "لغة الآلة" أو "لغة التجميع".

التنقلية portability: قابلية البرنامج للعمل على أكثر من نوع من الحواسيب.

تفسير interpret: تنفيذ سطور البرنامج المكتوب بلغة عالية المستوى واحدا بعد الاخر.

جمع البرنامج compile: ترجمة البرنامج المكتوب بلغة عالية المستوى إلى لغة منخفضة المستوى دفعة واحدة، استعدادا لتنفيذه لاحقا.

النص الاصلي source code: البرنامج المكتوب بعغة عالية المستوى قبل جمعه.

نص الكائن object code: ما ينتج عن جمع البرنامج.

قابل للتنفيذ executable: اسم اخر لنص الكائن عندما يكون جاهز للتنفيذ.

محث prompt: اشارات>>> يظهرها المفسّر، مظهرا استعداده لاستقبال المدخلات.

مخطوط برمجي script: برنامج خُزّن في ملف (عادة ما سيترجم لاحقا).

الوضع التفاعلي interactive mode: احد استخدامات مفسّر بايثون، بأن تطبع الاوامر و التعبيرات مباشرة بعد المحث.

الوضع الكتابي script mode: احد استخدامات مفسّر بايثون، بأن يقرأ و ينفذ عبارات نص برمجي.

برنامج program: مجموعة من التعليمات تحدد العملية الحوسبية.

خوارزمية algorithm: طريقة عامة لحل نوع من المشاكل.

بقة bug: خطأ في البرنامج.

تصحيح الاخطاء debugging: عملية تصيد أي من الأخطاء الثلاثة و تصحيحها.

النحو syntax: بناء البرنامج.

خطأ نحوي syntax error: خطأ في البرنامج، إما أملائي او نحوي، لا يمكن معه اعرابه (و بالتالي لا يمكن تفسيره).

استثناء exception: خطأ ظهر فقط عندما شغل البرنامج.

دلالة semantics: المقصود من البرنامج.

خطأ دلالي semantic error: خطأ في البرنامج يجعل ناتج البرنامج غير ذلك الذي قصد منه.

اللغات الطبيعية natural languages: تلك اللغات التي يتحدثها البشر، تطورت طبيعيا.

اللغات الرسمية formal languages: تلك اللغات التي صممها البشر لتستخدم في حالات محددة.

رمز token: أحد مكونات بناء نص البرنامج الاساسية، مرادف للكلمة في اللغات الطبيعية.

اعراب parsing: فحص نص البرنامج و تحليل بنائه النصي.

عبارة اطبع print statement: من تعليمات مفسّر بايثون، تظهر قيمة على الشاشة.

1.8 تمارين:

تمرين 1.2 اذهب إلى `http://python.Org`، تحتوي هذه الصفحة على معلومات عن بايثون و بها وصلات تتعلق به. تستطيع البحث هناك في وثائق بايثون.

مثلا ان بحثت عن كلمة `print` فان اول نتيجة ستظهر هي وثائق `print`. في هذه المرحلة قد لا تعني لك شيئا، إلّا ان معرفة مكانها سيكون مفيدا.

تمرين 1.3 شغل مفسِّر بايثون و اطبع `help()` لتشغِّل المساعدة، بامكانك طباعة:

```
help('print')
```

ان لم يعمل هذا المثال سيكون عليك تنصيب مزيدا من وثائق بايثون أو تحديد متغير البيئة، تفاصيل حل هذه المشكلة تعتمد على نظام التشغيل.

تمرين 1.4 شغل مفسِّر بايثون و استخدم البرنامج كحاسبة. بناء المعادلات في بايثون تقريبا نفس بناء المعادلات العادي، فمثلا الرموز + و - و / تعني الجمع و الطرح و القسمة الا ان رمز الضرب هو *.

ان ركضت 10 كم في سباق بمدة 43 دقيقة و 30 ثانية، فما هو معدل الوقت بالميل؟ ما هو معدل سرعتك بالميل في الساعة؟ (تلميح: يوجد 1.61 كم في الميل).

الفصل الثاني

المتغيرات، التعبيرات و العبارات

2.1 القيم و انماطها

احد الاشياء الاساسية التي يتعامل بها البرنامج هو **القيمة**، كالحرف أو الرقم، القيم التي مرت بنا حتى الان هي 1 و 2 و `'Hello, World!'`

تنتمي القيم لعدة أنماط: فـ 2 عدد صحيح، أما `'Hello, World!'` فهو سلسلة من الحروف (محارف). بإمكانك (و كذلك بإمكان المفسِّر) التعرف على المحارف لأنها محصورة بين علامتي اقتباس.

ان لم تكن متأكدا من نمط القيمة التي بين يديك فسيساعدك المفسِّر:

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

من الواضح لماذا يجب أن تنتمي المحارف **strings** إلى نمط اسمه `str` و أن تنتمي الاعداد الصحيحة **integers** إلى النمط `int`. الاقل وضوحا هو ان الارقام التي تحتوي على كسور عشرية (الاعداد الحقيقية) تنتمي إلى نمط يدعى (عائم) `float` و ذلك لأن هذه الارقام تمثَّل بصياغة تدعى **floating-point** (النقطة العشرية العائمة).

```
>>> type(3.2)
<type 'float'>
```

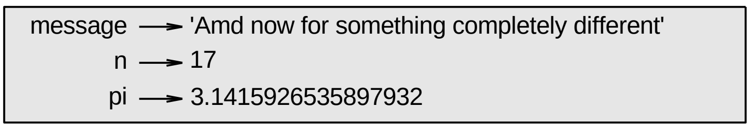
ماذا عن قيم كـ '17' و '32'؟ تبدو أرقاما، الا أنها بين علامتي اقتباس كالمحارف **strings**.

```
>>> type('17')
<type 'str'>
>>> type('32')
<type 'str'>
```

اذن فهي محارف.

عند كتابة الاعداد الكبيرة، يقسمها البعض إلى مجموعات من ثلاث خانات كـ 1,000,000 باستخدام فاصلة، هذا غير قانوني كعدد صحيح في بايثون، الا انه قانوني:

```
>>> 1,000,000
(1,0,0)
```



الشكل 2.1: رسم الحالة.

لم نتوقع هذا بالمرة!! لقد فسر بايثون الرقم 1,000,000 على أنه تسلسل أعداد صحيحة مقسمة بفاصلة. هذا أول مثال نراه على الأخطاءالدلالية: لم تظهر رسائل وجود الاخطاء، لكن البرنامج لا يفعل "ما قصد منه".

2.2 المتغيرات

من أقوى خصائص لغات البرمجة هي قدرتها على التلاعب **بالمتغيرات**، المتغير هو اسم يعطى لقيمة.

عبارات التعيين توجِد متغيرات جديدة و تعطيها قيم:

```
>>> message = 'And now for something completely different'
>>> n=17
>>> pi=3.1415926535897932
```

قام هذا المثال بثلاثة تعيينات، الاول تعيين محارف لمتغير اسمه `message` و الثاني أعطى عددا صحيحا للمتغير `n` و الثالث عَيَّن قيمة **π** (تقريبية) لـ `pi`.

الطريقة الشائعة عند تمثيل المتغيرات على الورق هي كتابة اسم المتغير ثم رسم سهم يؤشر لقيمته. هذا النوع من الأشكال يسمى رسم الحالة state diagram لأنه يظهر الحالة التي يوجد فيها المتغير (فكر بها كحالة المتغير الذهنية). الشكل 2.1 يوضح نتيجة المثال السابق.

نمط المتغير هو نمط القيمة المعينة له.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

2.3 تسمية المتغيرات و الاسماء المحجوزة

يختار المبرمجون لمتغيراتهم اسماء لها معنى في العادة، أي أنهم يوثِّقون لأي شيء سيستخدم ذلك المتغير منذ لحظة إيجاده.

يمكن لأسماء المتغيرات أن تكون طويلة، و ان تحوي حروفا و أرقاما، لكن يجب ان تبدأ بحرف. من المسموح استخدام الحروف الكبيرة، إلّا انه من الافضل استهلال اسم المتغير بحرف صغير (سترى لماذا مع سياق الحديث).

لا يمكن استخدام الفراغات أيضا.

يمكن لرمز الخط السفلي ،_، أن يظهر في اسم المتغير، يستخدم هذا الرمز عادةً لتقسيم اسماء المتغيرات المكونة من أكثر من كلمة لتعويض الفراغات، كـ `my_name` أو `air_speed_of_unladen_swallow`.

ان ستخدمت اسما ليس قانونيا لمتغير ستحصل على رسالة وجود خطأ نحوي:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
```

```
>>> more@=1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

كانت 76trombones غير قانونية لأن الاسم استهل بغير حرف، أما more@ فمشكلتها احتوائها على رمز غير قانوني@ لكن ما مشكلة class؟

سيتضح لنا بأن class هي **كلمة مفتاحية محجوزة**. يستخدم المفسّر كلمات مفتاحية لكي يتعرف على بناء البرنامج، و هذه الكلمات لا يجوز استخدامها لتسمية المتغيرات.

لدى بايثون2 إحدى و ثلاثين كلمة مفتاحية:

```
and       del      from     not      while
as        elif     global   or       with
assert    else     if       pass     yield
break     except   import   print
class     exec     in       raise
continue  finally  is       return
def       for      lambda   try
```

في بايثون 3 لم تعد exec كلمة مفتاحية، لكن nonlocal اصبحت كلمة مفتاحية.

ربما من الافضل لك ان تبقي هذه القائمة قريبة منك، فان اشتكى المفسِّر من استخدامك اسم ما لمتغير، ستعلم لماذا.

2.4 العوامل و مؤثراتها

المؤثرات هي رموز تمثل عمليات حوسبة كالجمع و الضرب اما القيم المُؤَثَّر عليها فهي العوامل.

المؤثرات +،-،/ و * تقوم بالجمع و الطرح و القسمة و الضرب كما في المثال التالي:

```
20+32    hour-1    hour*60+minutes    5**2    (5+9)*(15-7)
```

في لغات اخرى تستخدم اشارة ^ للتعبير عن الأُس، لكن في بايثون تستخدم هذه الاشارة في العمليات المنطقية و تدعى XOR. لن اشرح البتوايز (bitwise تعني المعالجة عن طريق اصغر جزء من المعلومة) هنا بامكانك القراءة عنها هنا http://wikipythonorg/moin/BitwiseOperators

في بايثون2 ، قد لا يقوم مؤثر القسمة بما تتوقع منه:

```
>>> minute=59
>>> minute/60
0
```

قيمة المتغير minute هي 59 و قسمتها على 60 يجب ان تنتج 0.98333 و ليس صفرا. السبب في هذا الخلل هو أن بايثون يقوم بقسمة أرضية (أي أن الناتج هو العدد بعد حذف الكسور 5/2=2) و عندما يكون طرفي القسمة اعداد صحيحة فالناتج سيكون أيضا عدد صحيح، القسمة الأرضية تحذف الكسور من ناتج القسمة و تقربه إلى الصحيح الاقل منه: كان هذا الصفر هنا.

في بايثون3، ناتج هذه القسمة / هو قيمة حقيقية (تقبل الكسور) و هنالك رمز جديد هو // يقوم بقسمة أرضية.

ان كانت قيمة أي من العاملين حقيقية، فان بايثون يقوم بقسمة حقيقية، و تكون نتيجتها حقيقية أيضا:

```
>>> minute/60.0
0.98333333333328
```

2.5 التعبيرات و العبارات

التعبير هو تركيبة من القيم و المتغيرات و المؤثرات. القيمة بحد ذاتها تعتبر تعبيرا، فكل ما يلي يعتبر تعبيرا جائزا (على اعتبار أن قيمةً ما عُينت للمتغير x):

```
17
x
x+17
```

العبارة هي لبِنة برمجية في النص يمكن لبايثون أن ينفذها، مرت علينا عبارتان `print` و عبارة التعيين.

عمليا، فالتعبير هو أيضا عبارة، لكن قد يكون من الاسهل التفكير بهما كشيئين مختلفين، الفرق الأهم هو أن للتعبير قيمة أما العبارة فلا.

2.6 وضع البرمجة التفاعلي و وضع كتابة نص البرنامج

من حسنات البرمجة باستخدام المفسّر هي انه يمكّنك من تجريب أجزاء البرنامج الصغيرة قبل وضعها في المخطوط البرمجي، إلا أن هناك فروقا بين وضعي استخدامه قد تلتبس عليك.

مثلا ان كنت تستخدم بايثون كآلة حاسبة، فقد تطبع:

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

السطر الاول عبارة تعين قيمة للمتغير `miles`، لكن ليس لهذا التعيين تأثير ظاهر. السطر التالي هو تعبير، لذا يقوم المفسّر بتقييمه و عرض الناتج، فنفهم عندها بأن الماراثون هو حوالي 42 كم.

إلا إنك ان طبعت نفس النص البرمجي في وضع كتابة النص البرمجي و شغلت البرنامج، فلن تحصل على شيء، لأن التعبير بحد ذاته ليس له تأثير مرئي في وضع كتابة النص البرمجي.

في الحقيقة، بايثون يقيم التعبير إلا أنه لا يظهر نتيجته إلا ان طلب منه ذلك:

```
miles = 2.62
print miles * 1.61
```

قد يكون هذا التصرف مشوشا في البداية.

في العادة و عندما يحتوي النص البرمجي على تسلسل من العبارات، فإن ناتج العبارات يظهر واحدا تلو الاخر كلما نفذت عبارة.

مثلا النص:

```
print 1
x = 2
print x
```

سيظهر المخرجات:

```
1
2
```

عبارة التعيين `x = 2` لم تنتج أي مخرجات.

تمرين 2.1 إطبع العبارات التالية في مفسِّر بايثون لتر ماذا تفعل:

```
5
x = 5
x + 1
```

2.7 تراتب عمليات المؤثرات

عندما يحتوي تعبير ما على أكثر من مؤثر فإن تراتب تقييم العمليات يعتمد على **قوانين الاولوية للعمليات الحسابية**. يتّبع بايثون الإجماع المعروف. الاختصار PEMDAS طريقة سهلة لتذكر هذا الاجماع:

- **P**arentheses: الاقواس، لها الاولية القسوى، و بإمكانك استخدامها لإرغام بايثون على اتباع التسلسل الذي تريده. فبما أن التعبيرات الموجودة بين أقواس تنفذ أولا فإن `2*(3-1)` = 4، و كذلك `(1+1)**(5-2)` تساوي 8. بإمكانك أيضا استخدام الاقواس لتسهيل قراءة العبارات، كما في `(minute*100)/60` ، حتى و إن لم تغير النتيجة.
- **E**xponentiation: الأسس هي التالية في الاولوية، فإن `2**1+1` تساوي 3 و ليس 4، و كذلك `3*1**3` تساوي 3 و ليس 27.
- **M**ultiplication و **D**ivision: الضرب و القسمة لهما نفس الاولوية، التي هي أعلى من الجمع و الطرح، و اللتان لهما نفس الأولوية كذلك فإن `2*3-1` تساوي 5 و ليس 4، كذلك `6+4/2` تساوي 8 و ليس 5.
- المؤثرات التي لها نفس الاولوية تنفذ من اليسار إلى اليمين (ما عدا الأس). لذا ففي التعبير `degrees/2*pi` تنفذ القسمة أولا و نتيجتها تضرب في `pi` . إن أردت القسمة على 2π فعليك استخدام الاقواس أو طباعة التعبير هكذا: `degrees/2/pi`.

شخصيا لا أجهد نفسي بتذكر أولوية باقي المؤثرات، فإن لم أكن واثقا استعملت الاقواس.

2.8 عمليات المحارف

في العموم، لا عمليات حسابية تطبق على المحارف، حتى و إن بدا الحرف كرقم، فكل ما يلي غير قانوني:

```
'2'-'1' 'eggs'/'easy'   'third'*'a charm'
```

المؤثر + يستخدم مع المحارف، لكنه لا يعمل كما تظن: بل يقوم بالإضافة، أي أنه يجمع محارفين بوصلهما طرفا لطرف كالتالي:

```
first = 'throat'
second = 'warbler'
print first + second
```

ناتج هذا البرنامج هو `throutwarbler`.

المؤثر * يعمل أيضا على المحارف، إلّا أنه يقوم بالتكرار فمثلا `'spam' * 3` ستصبح `'spamspamspam'` عند استخدام المؤثر * فإن واحدا من عاملي العملية يجب أن يكون عددا صحيحا.

تشابُه استخدام المؤثران * و + على المحارف و على الأعداد مبرر، فكما أن `3*4` تساوي `4+4+4` ، سنتوقع أيضا أن `spam*3` ستكرر `spam` ثلاث مرات أي `'spam' + 'spam' + 'spam'` ، و هو ما نتج. بالمقابل هناك فرق كبير بين التكرار و الإضافة للمحارف و بين الجمع و الضرب للأعداد، هل تستطيع التفكير بخاصية للجمع لا تشبه

الإضافة؟

2.9 الحواشي

كلما تضخم البرنامج و ازداد تعقيدا أصبحت قراءته و فهمه أصعب. فاللغات الرسمية كثيفة، و غالبا لا يكفي مجرد النظر إلى نص برمجي لفهم ما هو و ما الغاية منه.

لهذا السبب، فإن إضافة تعليقات إلى البرنامج، تفسّر بلغة طبيعية ما الذي يفعله، تعتبر فكرة جيدة. هذه التعليقات تسمى **حواشي**، و تستهل بالرمز #:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

في هذه الحالة ظهرت الحاشية في سطر لوحدها. بإمكانك وضع الحاشية في نهاية السطر البرمجي:

```
percentage = (minute * 100) / 60     #percentage of an hour
```

كل ما بعد # و إلى اخر السطر مهمل – لن يؤثر في سير البرنامج.

فائدة الحواشي تكمن في تفسيرها لخصائص البرنامج المبهمة، فافتراض أن من يقرأ البرنامج سيعرف "ما" الذي يقوم به مفهوم ، المفيد هو تفسير "لماذا" يقوم به.

الحاشية التالية ثرثرة و لا فائدة لها:

```
v = 5    # assign 5 to v
```

الحاشية التالية تحتوي على معلومات مفيدة و ليست ظاهرة في النص البرمجي:

```
v = 5    # velocity in meters/second
```

تسمية المتغيرات الحكيمة تقلل من الحاجة إلى الحواشي، في المقابل فالاسماء الطويلة تجعل التعبيرات المعقدة عصية على الفهم لذا فهنالك حاجة للمفاصلة.

2.10 علاج الاخطاء

في هذه المرحلة ستكون الأخطاء الكتابية التي ستقترفها في الأغلب أسماء المتغيرات الغير جائزة كـ `class` و `yield` التي هي كلمات محجوزة، أو كـ `odd~job` و `us$` تحتوي على حروف غير قانونية.

المشكلة التي يراها بايثون عندما تضع فراغا بين كلمتين من اسم المتغير، هي أنه يظن أنهما عاملين بدون مؤثر بينهما:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

عندما تظهر رسائل وجود خطأ نحوي، فإن الرسالة بحد ذاتها لا تدل على الخطأ نفسه، و لا تساعد كثيرا في حله. أكثر الرسائل شيوعا هي: `SyntaxError: invalid syntax` و `SyntaxError: invalid token` وكلاهما ليس مفيدا جدا.

رسائل الخطأ التي تظهر وقت التشغيل على الاغلب تقول "استُخدِم قبل التعريف"، مما يعني أنك تحاول استخدام متغيرٍ لم تعَيَّن له قيمة بعد، هذا الخطأ سيحدث ايضا عندما تخطئ تهجة اسم المتغير:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

اسماء المتغيرات تتغير بتغير حالة الحروف (كبيرة ام صغيرة)، فـ LaTeX ليست Latex

أكثر الأخطاء الدلالية التي سترتكبها في هذه المرحلة على الارجح ستكون تراتب العمليات. مثلا، عندما تحاول تقييم $\frac{1}{2\pi}$ قد تكتب التعبير هكذا:

```
>>> 1.0 / 2.0 * pi
```

و بما أن القسمة هي الاولوية هنا، فحاصل هذه العملية سيكون $\pi/2$ و هو ليس ما قصدت! بايثون لا يعلم ما هو قصدك، و لا يعلم أنَّ هذه النتيجة تختلف عما توقعت، لذلك لن يصدر أي رسالة تفيد بوجود خطأ، فقط ستحصل على إجابة خاطئة.

2.11 المعاني

قيمة value: من وحدات البيانات الاساسية، كالرقم و المحارف، التي يمكن لبرنامج استخدامها.

نمط type: فئة من القيم. الأنماط التي رأينها حتى الان كانت أعداد صحيحة (نمط int) و أعداد النقطة العائمة أو الاعداد الحقيقية (نمط float) و المحارف (نمط str).

عدد صحيح integer: نمط يمثل الاعداد بدون كسور.

نقطة-عائمة floating-point: نمط يمثل الاعداد مع كسورها.

محارف string: نمط يمثل تسلسل من الحروف و الارقام و الرموز القانونية.

متغير variable: أسم يشير لقيمة.

عبارة statement: جزء من النص البرمجي يمثل أمر أو فعل. العبارات التي مرت بنا حتى الان كانت تعيينات و print.

تعيين assignment: أي عبارة تعين قيمة لمتغير.

رسم الحالة state diagram: رسم بياني يمثل مجموعة من المتغيرات و القيم التي تؤشر إليها.

كلمة مفتاحية keyword: أسماء محجوزة، المفسِّر فقط يستطيع استخدامها للتعرف على بناء البرنامج، و لا يمكنك استخدامها لتسمية المتغيرات، الاسماء المحجوز كـ def, if, while و غيرها.

المؤثر operator: رمز يمثل حوسبة بسيطة كالجمع و الضرب أو إضافة المحارف.

العامل operand: إحدى القيم التي يؤثر فيها مؤثر العملية.

قسمة أرضية floor division: القسمة التي تحتفظ بأكبر عدد صحيح من الناتج و تُسقط الباقي.

تعبير expression: تركيبة من المتغيرات و المؤثرات و القيم، محصلته قيمة وحيدة.

تقييم evaluate: تبسيط التعبير عن طريق القيام بالعمليات لكي نحصل على قيمة وحيدة.

قوانين الاولوية rules of precedence: مجموعة القوانين التي تحكم الترتيب الذي ستنفذ بها عمليات فيها مجموعة من العوامل و المؤثرات لكي نحصل على النتيجة النهائية.

الاضافة concatenate: جمع المحارف طرف لطرف.

حاشية comment: معلومة عن البرنامج موجهة للمبرمجين (أو من يقرأ النص البرمجي) لا يكون لها تأثير على سير البرنامج.

2.12 تمارين

تمرين 2.2 إفترض أننا نفّذنا عبارة التعيين الاتية:

```
width = 17
height = 12.0
delimiter = '.'
```

اكتب القيمة و النمط (نمط قيمة التعبير) لكل من التعبيرات التالية:

1. width/2
2. width/20
3. height/3
4. 1 + 2 * 5
5. Delimiter * 5

استخدم مفسِّر بايثون للتأكد من اجاباتك.

تمرين 2.3 تدرب على استخدام مفسِّر بايثون كحاسبة.

1. حجم كرة نصف قطرها r هو $\frac{4}{3}r^3\pi$ ، فما هو حجم كرة نصف قطرها 5 (تلميح: حجمها ليس 392.7)
2. افترض أن سعر كتاب للمستهلك هو 24.95$، الا ان المكتبة تحصل على تخفيض 40%، رسوم الشحن 3$ لأول نسخة ثم 75 سنت لكل نسخة اضافية فما هي التكلفة الكلية لـ 60 نسخة؟
3. إن غادرت منزلي الساعة 6:52 صباحا للركض، فركضت أول ميل بسرعة منخفضة (8:15 للميل) ثم ثلاثة أميال بسرعة أعلى (7:12 للميل) ثم ميلا اخر بسرعة منخفضة، فكم تكون الساعة عند عودتي إلى البيت للافطار؟

الفصل الثالث

الاقترانات

3.1 نداء الاقترانات

في السياق البرمجي فإن الاقتران هو تسلسل لعبارات تقوم بعملية حوسبة و يكون له اسم. عندما تُعَرِّف اقتران، فإنك تعرف اسمه و تسلسل العبارات. ثم فيما بعد تنادي الاقتران باسمه. لقد مر علينا مثال لنداء اقتران:

```
>>> type(32)
<type 'int'>
```

اسم هذا الاقتران كان type. التعبير الموجود بين القوسين يسمى القرينة (قرينة الاقتران)، النتيجة (لهذه الاقتران) كانت نوع القرينة.

القول الشائع هو إن الاقتران "يأخذ" قرينة أو إنه "يُرجع" نتيجة. النتيجة تدعى القيمة المُرجَعة.

3.2 اقترانات تغيير الانماط

لدى بايثون اقترانات جاهزة تحول القيم من نمط إلى نمط اخر. اقتران int يأخذ أي قيمة و يحولها إلى عدد صحيح ان استطاع، و إلا فسيتذمر:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

بإمكان int تحويل قيم الأعداد الحقيقية إلى اعداد صحيحة. لكنه لا يقرب النتيجة، بل يحذف الكسور:

```
>>> int(3.9999)
3
>>> int(-2.3)
-2
```

أما float فيحول الاعداد الصحيحة و المحارف إلى أعداد حقيقية:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

و أخيرا str يحول قرينته إلى محارف:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.3 الاقترانات الرياضية

لدى بايثون **مديول** للرياضيات به معظم الاقترانات الرياضية المعروفة. المديول **module** هو ملف يحتوي على اقترانات عديدة تكون ذات صلة.

قبل ان نتمكن من استخدام المديول علينا استيراده:

```
>>> import math
```

هذه العبارة تستدعي المديول و تضيف لبرنامجك **كائن مُديول** اسمه `math`، ان طبعت المُديول ستظهر لك معلومات عنه:

```
>>> print math
<module 'math' (built-in)>
```

يحتوي كائن المديول على الاقترانات و المتغيرات المعرَّفة في المديول، لكي تَستخدم أحد الاقترانات عليك تحديد اسم المديول و كذلك اسم الاقتران مفصولان بنقطة (period) هذه الصياغة تدعى التنويت بالنقاط **dot notation** .

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

المثال الاول يستخدم اقتران `log10` لحساب نسبة `signal_to_noise` بالدسبل (على فرض أننا عرَّفنا `signal_power` و `noise_power`) . مديول `math` أيضا به `log` الذي تحسب لوغارتمات القاعدة e.

المثال الثاني يوجد جيب الزاوية الدائرية، تعمّدت جعل اسم المتغير (`radians`) للتذكير بأن اقترانات حساب المثلثات (`cos, tan` الخ) تعتمد الزوايا الدائرية. لتحول الزاوية من درجات إلى دائري، قسِّم القيمة على **360** ثم اضرب الناتج بـ 2ط:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
0.707106781187
```

3.4 التركيب

ما رأيناه حتى هذه الآن هو عناصر البرنامج، من متغيرات و تعابير و عبارات، منعزلة، دون التحدث عن طُرق الجمع بينها.

من اكثر وظائف لغات البرمجة فائدة هي قدرتها على أخذ لبنات بناء صغيرة و تركيبها (compose). فقرينة الاقتران على سبيل المثال قد تكون أي من التعبيرات، حتى أن المؤثرات الحسابية قد تصبح قرائن:

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

و قد يكون نداء اقتران بحد ذاته قرينة:

```
x = math.exp(math.log(x+1))
```

يمكنك وضع قيمة في أي موقع تقريبا، يمكنك وضع تعبير بالغ التعقيد، باستثناء وحيد: الطرف الايسر لعبارة تعيين يجب أن يكون اسم المتغير، أي تعبير اخر على يسار التعيين سيعتبر خطأً نحويا (سنرى الاستثناءات في هذا المضمار لاحقا).

```
>>> minutes = hours * 60          #right
>>> hours * 60 = minutes          #wrong!
SyntaxError: can't assign to operator
```

3.5 إضافة اقترانات جديدة

ما استخدمناه لحد الان هي اقترانات جاهزة أتت مع بايثون، الا انه بإمكاننا عمل اقترانات جديدة تضاف إلى البرنامج. **تعريف الاقتران** يحدد اسم الاقتران الجديد و العبارات التي ستنفذ عند نداء هذا الاقتران.

هذا مثال:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay"
    print "I sleep all night and I work all day"
```

الكلمة المفتاحية def تشير إلى ان هذا تعريف لاقتران، اسم الاقتران print_lyrics . قوانين تسمية الاقترانات هي نفسها قوانين تسمية المتغيرات: حروف و أرقام و بعض العلامات القانونية الاخرى، و أيضا يجب الا يستهل الاسم برقم و لا يجوز استخدام الكلمات المفاتيح كأسماء للاقترانات، و الأفضل تجنب استخدام نفس الإسم لمتغير و لاقتران.

القوسان الفارعان () بعد الاسم تشيران إلى ان هذا الإقتران لا يأخذ قرينة.

السطر الاول من تعريف الاقتران يسمى **الترويسة** header ، و باقي التعريف يسمى المتن (body) . يجب أن تنتهي الترويسة بنقطتان :. يجب أن تكون هناك مسافة بادئة لسطور متن الاقتران، اجمع المبرمجون أن تكون أربعة فراغات (انظر القسم 3.14). يمكن لمتن الاقتران ان يحتوي على أي عدد من العبارات.

في هذا الاقتران كانت المحارف محاطة بعلامات اقتباس مزدوجة " ، العلامة المزدوجة و العلامة المفردة تقومان بنفس الشيء، أغلب الناس يستخدمون العلامة المفردة، إلا إن تطلب الامر استخدام المزدوجة، فالعلامة المفردة هي أيضا علامة اختصار بالانجليزية كـ (can't = cannot). فإن احتوت الجملة المطبوعة على علامة اختصار تُستعمل العلامة المزدوجة.

ان طَبعت تعريف اقتران في الوضع التفاعلي فإن المفسِّر سيطبع نقاط المحذوف (ثلاث نقاط متتالية كتلك التي في "املأ الفراغ") لكي ينبهك إلى ان الاقتران غير مكتمل:

```
>>> def print_lyrics():
...     print "I'am lumberjack, and I'm okay"
...     print "I sleep all night and I work all day"
...
```

في الوضع التفاعلي فقط يجب ادخال سطر فارغ لكي تختم الاقتران، و ليس ضروريا في وضع كتابة النص.

تعريف الاقتران يوجد متغيرا بنفس الاسم:

```
>>> print print_lyrics
<function print_lyrics at 0x6d88ce9a>
>>> type(print_lyrics)
type 'function'
```

قيمة print_lyrics هي **كائن اقتران**، و لها النمط اقتران (function).

نحوُ كتابة نداء الاقتران الجديد هو نفس نحو الاقترانات الجاهزة:

```
>>> print_lyrics()
I'm lumberjack, and I'm okay
I sleep all night and I work all day.
```

بمجرد تعريفك لاقتران، يمكنك استخدامه من داخل اقتران آخر. فمثلا إن اردنا تكرار الجملتين السابقتين نستطيع كتابة اقتران جديد ينادي اقتراننا الحالي مرتين:

```
def repeat_lyrics():
```

```
    print_lyrics()
    print_lyrics()
```

ثم نادي repeat_lyrics:

```
>>> repeat_lyrics()
I'm lumberjack, and I'm okay
I sleep all night and I work all day
I'm lumberjack, and I'm okay
I sleep all night and I work all day
```

في الواقع لا تغنى هذه الاغنية هكذا.

3.6 التعريفات و استخداماتها

ان جمعنا فتات النص البرمجي من القسم السابق فإن البرنامج الكلي سيبدو كالتالي:

```
def print_lyrics():
    print "I'am lumberjack, and I'm okay"
    print "I sleep all night and I work all day"

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

يحتوي هذا البرنامج على اقترانين: print_lyrics و repeat_lyrics ، تنفذ الاقترانات كما تنفذ العبارات الاخرى، لكن تأثيرها هو خلق كائن اقتران، العبارات الموجودة داخل الاقتران لا تنفذ الا إذا نودي الاقتران، تعريف الاقتران لا يولد أية مخرجات.

كما تتوقع، فلا يمكن تنفيذ اقتران قبل خلقه، بكلمات اخرى، يجب تنفيذ تعريف الاقتران قبل ندائه للمرة الاولى.

تمرين 3.1 في البرنامج السابق، انقل السطر الاخير إلى بداية النص البرمجي، بحيث تظهر عبارة نداء الاقتران قبل التعريفات ثم شغل البرنامج و لاحظ رسائل الاخطاء.

تمرين 3.2 انقل نداء الاقتران إلى مكانها الاصلي ثم انقل التعريف repeat_lyrics إلى ما قبل print_lyrics ماذا تتوقع أن يحدث عند تشغيل البرنامج؟

3.7 سريان التنفيذ

لكي تضمن بأن الاقتران معرف قبل استخدامه عليك ان تعرف الترتيب الذي يتم وفقه تنفيذ العبارات، و هو ما يسمى سريان التنفيذ flow of execution.

يبدأ التنفيذ دائما بأول عبارة في البرنامج، تنفذ العبارات واحدة تلو الاخرى من الأعلى إلى الأسفل.

تعريفات الاقترانات لا تؤثر في هذا الترتيب، لكن تذكر بأن العبارات الموجودة داخل الاقتران لن تنفذ قبل نداء الاقتران.

في ترتيب سريان البرنامج، يشبه نداء الاقتران التحويلة، فبدلا من الانتقال إلى العبارة التالية في الترتيب، فإن السريان سيقفز إلى متن الاقتران المنادى، و ينفذ كل العبارات هناك ثم يعود إلى الموضع الذي قفز منه.

قد يبدو هذا بسيطا إلى تتذكر بأن اقتران ما يمكنه نداء اخر، فبينما يكون البرنامج في وسط اقتران ما قد يتطلب نداء عبارة

من اقتران اخر، بل إنه قد يتطلب نداء اقتران ثالث بينما هو في الاقتران الثاني.

لحسن الحظ بايثون يتذكر جيدا من أين انطلق في القفزة الاولى، ففي كل مرة ينتهي من تنفيذ اقتران، يقفز عائدا إلى النقطة التي نودي الاقتران منها و يكمل، و عندما يصل إلى نهاية البرنامج ينقضي.

ما العبرة الاخلاقية من هذه الحكاية؟ هي أنه عندما تقرأ برنامجا، فأنت لست مضطرا لقراءة النص من الاعلى إلى الاسفل، بل من المنطقي أن تتبع ترتيب سريان البرنامج.

3.8 البرمترات و القرائن

بعض الاقترانات الجاهزة التي رأيناها تتطلب قرائن، فعندما تنادي math.sin عليك تمرير رقم ما كقرينة، بعض الاقترانات تتطلب أكثر من قرينة، math.pow تتطلب قرينتين مثلا.

داخل الاقتران، تُعَّين هذه القرائن لمتغيرات تدعى برمترات، هذا مثال لاقتران أوجده المستخدم يأخذ قرينة:

```
def print_twice(bruce):
   print bruce
   print bruce
```

هذا الاقتران يعين قرينة لبرمتر اسمه bruce . عندما ينادى فسيطبع قيمة البرمتر (بغض النظر ما هو) مرتين.

و هذا الاقتران يعمل مع أي قيمة تمكن طباعتها.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

نفس قوانين التركيب التي تنطبق على الاقترانات الجاهزة تنطبق أيضا على اقترانات المستخدم. فبإمكاننا استخدام أي نوع من العبارات كقرينة لـ print_twice:

```
>>> print_twice('Spam'*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

تُقيَّم القرينة قبل نداء الاقتران، ففي الامثلة السابقة أُوجدت قيم 'Spam' *4 و math.cos(math.pi) مرة واحدة.

يمكنك أيضا استخدام متغيرات كقرائن:

```
>>> michael = 'Eric, the half of bee.'
>>> print_twice(michael)
Eric, the half of bee.
Eric, the half of bee.
```

اسم المتغير الذي مررناه كقرينة للاقتران print_twice كان michael و لا علاقة له باسم البرمتر(bruce)، إذن لا يهم أي قيمة ناديناها لتعود إلى منزلها (في المنادي) . فنحن في هذه الاقتران ندعو كل الناس "أبو شريك" bruce.

3.9 المتغيرات و البرمترات موضعية

عندما توجِدُ متغيرا داخل اقتران، فإنه يكون موضعيا (محليا)، أي أنه يوجد فقط داخل الاقتران، مثلا:

```
def cat_twice(part1,part2):
    cat = part1 + part2
    print_twice(cat)
```

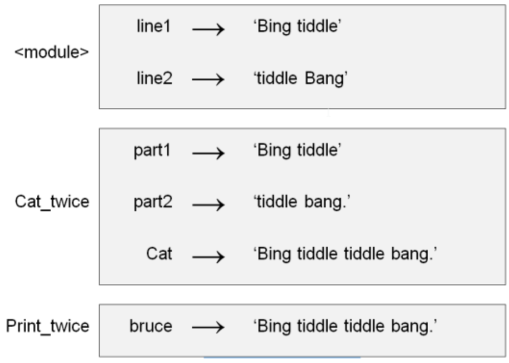
يأخذ هذا الاقتران قرينتين، يجمعها ثم يطبع النتيجة، هذا مثال لاستخدامه:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang
Bing tiddle tiddle bang
```

عندما تُغلَق `cat_twice` فإن `cat` سيتلف، و ان حاولنا طباعته سنحصل على استثناء:

```
>>> print cat
NameError: name 'cat' is not defined
```

البرمترات كذلك موضعية، فخارج حدود `print_twice` لا معنى لـ `bruce`.



الشكل 3.1: الرسم المستف

3.10 الرسم المستف: stack diagram

لتتمكن من ملاحقة أين المتغيرات و أماكن استخدامها، من المفيد أحيانا عمل رسم مستف. كما رسم الحالة، الرسم المستف يبين قيمة كل من المتغيرات، بالاضافة لذلك فهو يبين الاقتران التي ينتمي اليه ذلك المتغير.

يمثَّل كل اقتران بإطار، الاطار هو صندوق يكتب إلى جانبه اسم الاقتران، و بداخله اسماء برمترات و متغيرات الاقتران، الرسم المستف للمثال السابق يظهر في الشكل 1.3.

ترتب الاطارات في الرسم المستف على شكل تكديسة تبين أي اقتران استدعى أي اقتران اخر. في مثالنا هذا كان الذي استدعى `print_twice` هو `cat_twice`، أما `cat_twice` فقد استدعاه `__ main __`، و هو اسم مخصوص للاطار الموجود في قمة التستيفة، عندما تخلق متغيرا خارج أي اقتران فإنه ينتمي إلى `__ main __`.

كل برمتر في الاقتران يشير إلى نفس قيمة قرينته، فـ `part1` له نفس قيمة `line1`، كذلك `part2` و `line2` و لـ `bruce` نفس قيمة `cat`.

ان ظهر خطأ عند نداء الاقتران فسيطبع بايثون اسم الاقتران و الاقتران التي استدعاه، و اسم الاقتران الذي استدعى الاقتران الذي استدعاه، و هكذا حتى تصل إلى `__ main __`.

فمثلا عندما تحاول النفاذ إلى cat من داخل print_twice ستحصل على NameError:

```
Traceback (innermost last):
   file "testpy", line 12, in __main__
      cat_twice(line1, line2)
   file "testby", line 5, in cat_twice
      print_twice(cat)
   file "testpy", line 9, in print_twice
      print cat
NameError: name 'cat' is not defined
```

قائمة الاقترانات هذه تسمى "الملاحقة" traceback. وهي تقول لك في أي من ملفات البرنامج وقع الخطأ، و في أي سطر، و كذلك يبلغك بالاقترانات التي كانت تنفذ حين وقوع الخطأ، و تظهر كذلك السطر البرمجي الذي سبب الخطأ.

تراتب الاقترانات في الملاحقة هو نفس التراتب الموجود في الرسم المستف، فالاقتران المنشغِل حاليا يقبع في الاسفل.

3.11 الاقترانات المثمرة و الاقترانات العقيمة

بعض الاقترانات التي بين يدينا، كالاقترانات الحسابية، تؤتي بنتيجة، و لعدم وجود اسم أفضل سميتها "اقترانات مثمرة"، الاقترانات الاخرى كـ print_twice التي تقوم بعملية ما لكنها لا ترجع قيمة، هذه الاقترانات تسمى عقيمة **void**.

انت في أغلب الاحوال تنادي اقتران مثمر لكي تستخدم ثمرته، فمثلا قد تعين نتيجة الاقتران لمتغير ما أو تستخدم النتيجة في تعبير:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

عند نداء الاقتران في الوضع التفاعلي فإن بايثون سيظهر النتيجة:

```
>>> math.sqrt(5)
2.2360679774997898
```

لكن في وضع كتابة النص، عندما تنادي اقتران مثمر كما هو فإن نتيجته ستفقد إلى الابد!

```
Math.sqrt(5)
```

يقوم النص بحساب الجذر التربيعي لـ 5 ، لكن طالما أنها لم تطبع أو تخزن فإنها تصبح عديمة الفائدة.

قد تعرض الاقترانات العقيمة شيئا على الشاشة أو تقوم بتأثير ما، لكنها لا ترجع قيمة، فإن حاولت تعيين نتيجة اقتران عقيم لمتغير فستحصل على قيمة خاصة تسمى "عدم" None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

القيمة None ليست المحارف 'None' و هي أيضا قيمة لها نمطها الخاص **None**:

```
>>> print type(None)
<type 'NoneType'>
```

جميع الاقترانات التي كتبناها حتى الان هي اقترانات عقيمة، سنبدأ بكتابة اقترانات مثمرة بعد بضعة فصول.

3.12 الاقترانات، لماذا؟

قد لا تكون الفائدة من تحمل عناء تقسيم البرنامج إلى اقترانات جلية بعد. حسنا، هنالك عدة اسباب:

- عمل اقتران جديد يمكنك من تسمية عدة عبارات كمجموعة واحدة، مما يسهل قراءة و علاج اخطاء البرنامج.
- بإمكان الاقترانات تصغير حجم البرنامج، حيث أنها تقلل من تكرار النص البرمجي، ثم عندما تريد تعديل النص لاحقا، سيكون عليك تعديله في موضع واحد.
- تقسيم برنامج طويل إلى اقترانات يمكّنِك من علاج أخطاء الأجزاء واحدا بعد الاخر ثم تجميعها لتعمل في النهاية ككل.
- الاقترانات المصممة جيدا مفيدة لأكثر من برنامج، فبمجرد كتابتك لاقتران و علاجك لأخطائه تستطيع استخدامه في مكان اخر.

3.13 الاستيراد باستخدام from

يزودك بايثون بطريقتين لاستيراد المديولات، رأينا منها:

```
>>> import math
>>> print math
<module 'math' (built-in)>
>>> print math.pi
3.14159265359
```

عندما تستورد math فإنك توجد في برنامجك كائن مديول يسمى math يحتوي هذا الكائن على ثوابت كـ pi و اقترانات كـ sin و exp.

إلا انك عندما تحاول الوصول إلى pi بمفردها ستحصل على خطأ:

```
>>> print pi

Traceback (most recent call last):
File "<pyshell#0>", line 1, in <module>
print pi
NameError: name 'pi' is not defined
>>>
```

البديل هو أن تستورد pi فقط من مديول math :

```
>>> from math import pi
```

عندها يمكنك استخدام pi مباشرة:

```
>>> print pi
3.14159265359
```

أو أنك تستورد كل ما في المديول باستخدام الرمز *:

```
>>> from math import *
>>> cos(pi)
-1.0
```

فائدة استيراد كل شيء من المديول هي أن النص البرمجي يصبح موجزا، سيئتها هو احتمال وجود تشابه في الاسماء المعرَّفة في المديولات المختلفة، أو بين اسم يستعمله المديول و اسم تستعمله انت في برنامجك.

3.14 علاج الاخطاء

استخدام محرر نصوص عادي لكتابة النصوص البرمجية لبايثون قد يسبب الأخطاء، خصوصا بالنسبة للمسافات البادئة، افضل طريقة لتجنب هذه الأخطاء هو استخدام الفراغات المفردة و ليس فراغات الجدولة (TABS)، معظم محررات النصوص التي سمعت ببايثون تقوم بوضع المسافات اليا.

الفراغات و مسافات الجدولة غير مرئية، مما يصعب علاج الأخطاء الناجمة عنها، لذلك حاول العثور على محرر نصوص يقوم بها اليا.

لا تنس حفظ برنامجك قبل تشغيله، بعض المحررات تخزنه لك عندما تطلب تشغيله، الخطأ المحتمل هنا هو أنك شغلت برنامجا غير الذي تنظر اليه في محرر النصوص.

قد يستهلك علاج الأخطاء الكثير من الوقت عندما تعتقد بأن البرنامج الذي شغلته هو البرنامج الذي تنظر اليه في المحرر. فمهما عدلت في المحرر ستحصل على نفس الخطأ مرارا و تكرارا.

ان لم تكن متأكدا من أن النص البرمجي في المحرِّر هو الذي يعمل عندما تشغل البرنامج، إدرج عبارة في النص البرمجي كـ `print 'hello'` فإن لم تر `Hello` في البرنامج ستعلم بأنك تشغل برنامجا اخر.

3.15 المعاني

اقتران function: مجموعة من العبارات التي تنفذ عملية مفيدة ما، تكون تحت مسمى واحد. الاقترانات قد تأخذ قرائن أو قد لا تأخذ و قد تعود بنتيجة أو لا.

تعريف اقتران function definition: عبارة تنشئ اقترانا جديدا، تحدد له اسم و برمترات و كذلك عبارات ينفذها.

كائن الاقتران function object: قيمة أنتجها تعريف الاقتران. فاسم الاقتران هو متغير يشير إلى كائن الاقتران.

ترويسة header: أول سطر في تعريف الاقتران.

متن body: سلسلة العبارات الموجودة داخل تعريف الاقتران.

برمتر parameter: اسم يستخدم داخل الاقتران. يشير إلى القيمة التي مررت كقرينة.

نداء الاقتران function call: عبارة تنفذ الاقتران. تتألف من اسم الاقتران يتبعه سلسلة من القرائن.

قرينة argument: قيمة تمرر إلى الاقتران عند ندائه. و هذه القيمة تعيَّن للبرمتر المقصود.

متغير موضعي local variable: هو متغير عرِّف داخل اقتران. المتغير الموضعي يستعمل فقط داخل الاقتران.

القيمة المرجعة return value: نتيجة الاقتران. ان استُخدم نداء الاقتران كتعبير فإن القيمة المرجعة هي قيمة التعبير.

اقتران مثمر fruitful function: اقتران يعود بقيمة.

اقتران عقيم void function: اقتران لا يعود بقيمة.

مديول module: ملف يحتوي على تشكيلة من الاقترانات و غيرها من التعريفات. تكون من نفس الموضوع.

عبارة الاستيراد import statement: عبارة تقرأ ملف المديول و تخلق كائن مديول.

كائن مديول module object: القيمة التي أنشأتها عبارة الاستيراد `import`، تسمح هذه القيمة بالوصول إلى القيم الموجودة في المديول.

التنويت بالنقط dot notation: تركيبة اعراب تستخدم لنداء اقتران من داخل ملف مديول بأن تكتب اسم المديول تتبعها بنقطة ثم اسم الاقتران.

التركيب composition: استخدام تعبير كجزء من تعبير أكبر، أو عبارة كجزء من عبارة أكبر.

سريان التنفيذ flow of execution: ترتيب تنفيذ العبارات خلال تشغيل البرنامج.

الرسم المستف stack diagram: تمثيل صوري لرصة من الاقترانات و متغيراتها و القيم التي تشير اليها.

اطار frame: صندوق مرسوم في الرسم المستف يمثل نداءا لاقتران. تكتب فيه المتغيرات الموضعية و قيمها.

الملاحقة traceback: قائمة بالاقترانات التي تنفذ حاليا، تطبع هذه القائمة عند حدوث استثناء.

3.16 تمارين

تمرين 3.3: لدى بايثون اقترانا جاهزا يسمى `len` يُرجع طول المحارف، فقيمة `len('allen')` هي 5

اكتب اقترانا و سمه `right_justify` يأخذ محارف اسمها `s` كبرمتر ثم يطبع ما يكفي من الفراغات قبل المحارف `s` بحيث يكون اخر حرف من `s` في العمود 70 من الشاشة.

```
>>> right_justify('allen')
                                 allen
```

تمرين 4.3: كائن اقتران هي قيمة يمكنك تعيينها لمتغير او تمريرها كقرينة، مثلا `do_twice` هو اقتران يأخذ كائن اقتران كقرينة و يناديه مرتين:

```
def do_twice(f):
   f()
   f()
```

هذا مثال لاستخدام `do_twice` لنداء اقتران اسمه `print_spam` مرتين:

```
dif print_spam():
   print 'spam'
do_twice(print_spam)
```

1. اكتب المثال في نص برمجي ثم جربه.
2. عدل `do_twice` بحيث يأخذ قرينتين، كائن اقتران و قيمة و ينادي الاقتران مرتين ممررا القيمة كقرينة.
3. اكتب صيغة أعم من `print_spam` و سمها `print_twice` ، بحيث تأخذ محارف كبرمتر ثم تطبعها مرتين.
4. استخدم الصيغة المعدلة من `do_twice` بحيث تستدعي `print_twice` و تمرر لها `'spam'` كقرينة.
5. عرف اقترانا جديدا و سمه `do_four` بحيث يأخذ كائن اقتران و قيمة و ينادي الاقتران أربعة مرات، ممررا له القيمة كبرمتر على أن تكون في متن الاقتران عبارتين فقط و ليس اربعة.

الحلول: `http://thinkpython.com/code/do_four.py`.

تمرين 3.5 يمكنك حل هذا التمرين باستخدام العبارات و خصائص بايثون التي مرت علينا حتى الان:

اكتب اقترانا يرسم شبكة كهذه:

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

تلميح: لتطبع أكثر من قيمة على سطر واحد، اطبع تسلسلا مقسما بفاصلة:

```
Print '+' , '-'
```

عندما ينتهي التسلسل بفاصلة فإن بايثون يترك السطر غير منهي، و هكذا فالقيمة التي طبعت بعده تظهر على نفس السطر.

```
Print '+',
print '-'
```

ما سينتج من هذه العبارات هو `'+ -'` ، أما عبارة `print` لوحدها فستنهي السطر و تنتقل للسطر التالي.

اكتب اقترانا اخر يرسم نفس الشبكة لكن بأربعة صفوف و اربعة اعمدة.

الحلول: `http://thinkpython.com/code/grid.py`. **عرفان**: بني هذا التمرين على تمرين في كتاب والين، برمجة سي العملية، الاصدار الثالث، أوريلي ميديا، 1997.

الفصل الرابع

دراسة حالة: تصميم الواجهة

أمثلة النص البرمجي لهذا الفصل متوفرة على http://thinkpython.com/code/polygon.py

4.1 عالم السلحفاة

كتبت **رزمة** برمجية لترافق هذا الكتاب و سميتها Swampy، بامكانك تحميل سوَمبي من http://thinkpython.com/swampy ثم اتباع التعليمات هناك لتنصيبها في نظام التشغيل.

الرزمة هي مجموعة من المديولات، احدى المديولات في سومبي هي TurtleWorld توفر لك مجموعة من الاقترانات لترسم خطوطا عن طريق قيادة سلاحف على الشاشة.

ان كنت قد نصّبت سومبي كرزمة فبإمكانك استيرادها هكذا:

```
from swampy.TurtleWorld import *
```

اما إن كنت قد حملت مديولات سومبي و لم تنصبها كرزمة فبإمكانك إمّا استخدامها في المجلد الذي يحتوي ملفات سومبي، او اضافة ذلك المجلد إلى مسار بحث بايثون. عندها يمكنك استيراد TurtleWorld هكذا:

```
from TurtleWorld import *
```

كيفية تنصيب و اعداد مسار البحث لبايثون تعتمد على أنظمة التشغيل. و لئلا يزدحم الكتاب بتفاصيل غير بايثون، فضلت وضع تلك التعليمات حول سومبي و انظمة التشغيل العديدة هنا: http://thinkpython.com/swampy

انشئ ملفا جديدا و سمه mypolygon.py ثم اطبع النص البرمجي التالي:

```
from swampy.TurtleWorld import *

world = TurtleWorld()
bob = Turtle()
print bob

wait_for_user()
```

السطر الاول سيستورد كل ما في مديول TurtleWorld من رزمة swampy.

في السطور التالية نوجد اقتران تجلية TurtleWorld و نعينه لـ world ، ثم Turtle و نعينه لـ bob ، طباعة bob ستنتج شيئا كهذا:

```
<TurtleWorld.Turtle object at 0xb7bfbf4c>
```

هذا يعني بأن bob يشير إلى تجلية instance من السلحفاة كما هي معرَّفة في مديول TurtleWorld. و معنى instance هنا هو "عضو في مجموعة"، هذه السلحفاة هي واحدة من مجموعة السلاحف المتوفرة في المديول.

wait_for_user هنا تأمر عالم السلحفاة بأن ينتظر حتى يقوم المستخدم بشيء ما، نعلم أنْ ليس هناك الكثير ليقوم به المستخدم غير اغلاق النافذة.

يوفر عالم السلحفاة عدة اقترانات لقيادة السلحفاة: fd و bk إلى الامام و إلى الخلف،lt و rt إلى اليسار و إلى اليمين. كل سلحفاة تحمل قلما، اما ان يكون مرفوعا أو موضوعا، ان كان موضوعا فستترك السلحفاة أثرا عند سيرها. الاقترانات pu و pd تعني قلما مرفوعا و قلما موضوعا.

لترسم زاوية قائمة، اضف السطور التالية (بين سطر ايجاد bob و سطر wait_for_user):

```
fd(bob, 100)
lt(bob)
fd(bob, 100)
```

السطر الاول يأمر bob بالتقدم 100 خطوة، السطر التالي يأمره الالتفاف يسارا. عند تشغيل البرنامج. سترى bob يسير شرقا ثم شمالا، تاركا قطعتين مستقيمتين خلفه.

الان عدل البرنامج لترسم مربعا، لا تكمل من هنا حتى تتقنه!

4.2 تكرار بسيط

أغلب الظن انك كتبت شيئا كالتالي (بدون اظهار نصوص ايجاد السلحفاة و انتظار المستخدم):

```
fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
```

جيد، إلا انه يمكننا القيام بنفس الشيء لكن أوجز عن طريق عبارة for ، اضف المثال التالي لـ mypolygon.py ثم شغله ثانية:

```
for I in range(4):
   print 'Hello!'
```

سترى شيئا كهذا:

```
Hello!
Hello!
Hello!
Hello!
```

كان هذا ابسط استخدامات عبارة for، سنرى الكثير منها لاحقا، إلا أن هذا المثال كاف ليجعلك تشطب ما كتبت لرسم المربع! لا تنتقل إلى التالي حتى ترسم المربع باستخدام for.

هذه عبارة for التي ترسم مربعا:

```
for I in range(4):
   fd(bob, 100)
   lt(bob)
```

بناء عبارة `for` كبناء تعريفات الاقترانات فلها ترويسة تنتهي بنقطتين ثم المتن المحدد بمسافات بادئة. للمتن ان يحتوي على اي عدد من العبارت.

عبارة `for` تدعى احيانا حلقة loop لأن لأن تنفيذ العبارات يدور عبر متن العبارة ثم يعود إلى البداية، في حالتنا هذه فالدوران في الحلقة كان لأربعة مرات.

هذه النسخة تختلف عن سابقتها لرسم المربع، لأن آخر ما ينفَّذ هو الالتفاف إلى اليسار بعد رسم اخر ضلع للمربع. تأخذ هذه الالتفافة الاخيرة بعضا من وقت البرنامج، الا انها تبسط النص البرمجي ان كنا نقوم بنفس العمل كل دورة في الحلقة. و لهذه النسخة تأثير آخر، فهي تترك السلحفاة في موقع البداية ملتفتة إلى الاتجاه الاصلي.

4.3 تمارين

مايلي هو سلسلة تمارين باستخدام TurtleWorld . المطلوب منها هو ان تكون ممتعة، إلا ان لها هدف ايضا. بينما تعمل عليها فكر بما قد يكون هذا الهدف.

في الاقسام اللاحقة هنالك حلول لهذه التمارين، فلا تنظر هناك حتى تنتهي من حلها (او حاولت على الاقل).

1. اكتب اقترانا اسمه `square` يأخذ برمتر اسمه `t` ، الذي سيكون السلحفاة، على هذا الاقتران ان يرسم مربعا باستخدام السلحفاة.
   اكتب عبارة نداء اقتران بحيث تمرر bob كقرينة لـ `square` ، ثم شغل البرنامج مرة اخرى.
2. اضف لـ `square` برمتر اخر و سمه `length` ، عدل في متن الاقتران بحيث يكون طول الاضلاع هو `length` ثم عدل في عبارة نداء الاقتران بحيث تحتوي على قرينة ثانية، شغل البرنامج ثم جربه بمدى من القيم لـ `length`.
3. الاقترانات `lt` و `rt` تقوم بالالتفاف 90 درجة كإعداد ابتدائي، الا انه يمكنك اضافة قرينة تحدد بها درجات الالتفاف، مثلا: `lt(bob, 45)` ستدير bob 45 درجة إلى اليسار.

اعمل نسخة من `square` و سمها `polygon` اضف لها برمترا اخر اسمه `n` ثم عدل متن الاقتران بحيث يرسم `bob` مضلعا عدد اضلاعه n. تلميح: الزاوية الخارجية لمضلع n/360 درجة.

4. اكتب اقترانا اسمه `circle` يأخذ السلحفاة `t` و نصف القطر `r` كبرمترات، بحيث يرسم دائرة تقريبية عن طريق استدعاء `polygon` بعدد مناسب من الاضلاع و بتعديل اطوالها، جرب نصك بعدة قيم لـ `r`.
   تلميح: تذكر حساب محيط الدائرة ثم تأكد من ان `length * n` = المحيط.
   تلميح اخر: ان كنت ترى ان `bob` بطيء، سرّعه باستخدام `bob.delay` و هو الزمن بين كل حركة له بالثواني `Bob.delay = 0.01` لتحركه بسرعة.
5. اعمل نسخة معممة اكثر من `circle` و سمها `arc` ، تأخذ القرينة الإضافية `angle` التي تحدد مقدار الجزء الذي سيُرسم من الدائرة، `Angle` مقاسة بالدرجات، فعندما تكون `angle = 360` سيرسم `bob` دائرة كاملة.

4.4 الكبسلة

طلب منك في التمرين الاول وضع نص رسم المربع في تعريف اقتران ثم نداء الاقتران، بحيث تمرر السلحفاة كبرمتر، هنا حل

المثال:

```
def square(t):
   I in range(4):
      fd(t, 100)
      lt(t)
square(bob)
```

العبارات الموجودة في قلب البرنامج، lt و fd مسافاتها البادئة مضاعفة لتبين انها داخل حلقة for ، و التي بدورها داخل تعريف اقتران. السطر التالي ، square(bob) كتب على بداية السطر، أي أنه ينهي كل من حلقة for و تعريف الاقتران

داخل الاقتران، t تشير إلى نفس السلحفاة bob ، أي أن lt(t) تعمل كـ lt(bob) اذن فلماذا أتحمل العناء، لماذا لا أسمي البرمتر bob؟ الفكرة هي أن t قد تكون أي سلحفاة، و ليس فقط bob، هذا يمكنك من ايجاد سلحفاة ثانية و تمررها كقرينة لـ square:

```
ray = Turtle()
square(ray)
```

تغليف قطعة برمجية في اقتران يسمى كبسلة، من فوائد الكبسلة هو أنها تلحق اسما بالنص البرمجي، مما يخدمك في توثيق ما تكتبه، من الايجاز نداء اقتران مرتين بدلا من قص و نسخ متن الاقتران.

4.5 التعميم

الخطوة التالية هي اضافة البرمتر length إلى square هاك الحل:

```
def square(t, length):
   for I in range(4):
      fd(t, length)
      lt(t)
square(bob, 100)
```

اضافة برمتر لاقتران يسمى تعميما، لأنه يجعل الاقتران يستخدم لأكثر من وظيفة: في النسخة السابقة كان لأضلاع المربع نفس الطول، الطول في هذه النسخة قد يكون بأي مقدار.

الخطوة التي تلي هي أيضا تعميم، فبدلا من رسم مربع، polygon يرسم مضلعا منتظما له اي عدد من الاضلاع تريد. هاك الحل: القانون

```
def polygon(t, n, length):
   angle = 360.0/n
   for I in range(n):
      fd(t, length)
      lt(t, angle)
polygon(bob, 7, 70)
```

سيرسم هذا النص سباعيا طول اضلاعه 70 ، و ان كان لديك أكثر من بضعة قرائن عددية، فالاسهل نسيان ما هي، أو ما هو ترتيبها، من الجائز ، و احيانا من المفيد، تضمين البرمترات في قائمة القرائن:

```
polygon(bob, n=7, length=70)
```

تسمى هذه ***كلمات مفتاحية قرينية*** لأنها تتضمن اسماء البرمترات ككلمات مفاتيح (لا تخلط بينها و بين كلمات بايثون المفتاحية كـ while و def).

هذا التركيب يسهل قراءة البرنامج و ينبه كذلك إلى كيفية عمل القرائن و البرمترات: فعند نداء الاقتران تعين القرائن

للبرمترات.

4.6 تصميم الواجهة

الخطوة التالية هي أن تكتب circle ، التي تأخذ نصف القطر r كبرمتر هاك حل بسيط يستخدم polygon لرسم مضلعا خمسيني:

```
def circle(t, r):
   circumference = 2 * math.pi * r
   n = 50
   length = circumference / n
   polygon(t, n, length)
```

يحسب السطر الاول محيط الدائرة التي نصف قطرها r بالمعادلة 2 نق.ط، و بما أننا نستخدم math.pi فعلينا استيراد math. اجمع المبرمجون ان تكون عبارات import في بداية النص البرمجي.

n هي عدد القطع المستقيمة التي تكوِن دائرتنا التقريبية، اذن length يكون طول كل قطعة مستقيمة و هكذا سيرسم لنا polygon مضلعا خمسينيا يمثل تقريبا لدائرة نصف قطرها r.

هناك تقييد واحد لهذا الحل،هو أن n عدد ثابت، مما يعني أنه عندما نرسم دائرة كبيرة جدا سيكون طول القطع المستقيمة كبير، و عندما نرسم دائرة صغيرة جدا سنضيع الوقت برسم قطع مستقيمة صغيرة. الحل الاوحد لهذا التقييد هو تعميم الاقتران بأخذ n كبرمتر، و هكذا سنعطي حرية اكثر للمستخدم (من يستدعي circle أيا كان)، الا ان الواجهة ستكون اقل نظافة.

واجهة اقتران هي اجمال لكيفية استخدامه: ما هي البرمترات؟ ما الذي يفعله الاقتران؟ واجهة الاقتران تكون "نظيفة" ان كانت "بسيطة قدر الامكان، لكن ليس أبسط. (اينشتين)".

في هذا المثال، تنتمي r إلى الواجهة لأنها تحدد الدائرة التي سترسم. أما n فليست مناسبة لأنها تتبع الكيفية التي ستتجسد بها الدائرة .

الافضل،بدلا من إعمام الفوضى في الواجهة، اختيار قيمة مناسبة لـ n حسب طول محيط الدائرة:

```
def circle(t, r):
   circumference = 2 * math.pi * r
   n = int(circumference / 3) + 1
   length = circumference / n
   polygon(t, n, length)
```

هكذا سيصبح عدد القطع المستقيمة يساوي (تقريبا) ثلث محيط الدائرة فيكون طول القطع المستقيمة (تقريبا) 3 و هي صغيرة بما يكفي بحيث تظهر الدائرة كدائرة، و كبيرة بما يكفي بحيث تكون فعالة و مناسبة لأي حجم من الدوائر.

4.7 التفتيت و البناء

عندما كتبت circle كان بوسعي استخدام polygon مرة اخرى، لأن المضلع تقريب مقبول للدائرة، لكن الاقواس ليست متساهلة كالدائرة، فليس بوسعنا استخدام المضلع او الدائرة لرسم القوس.

حل بديل لهذا هو نسخ النص البرمجي لـ polygon و تحويله إلى نص لقوس، قد تكون النتيجة كالتالي:

```
def arc(t, r, angle):
   arc_length = 2 * math.pi * r * angle / 360
   n = int(arc_length / 3) + 1
   step_length = arc_length / n
   step_angle = float(angle) / n

   for I in range(n):
      fd(t, step_length)
      lt(t, step_angle)
```

النصف الثاني من هذا الاقتران يبدو كمضلع، الا اننا لن نستطيع استخدام polygon من دون تعديل الواجمة، قد نستطيع تعميم polygon ليأخذ الزاوية كقرينة ثالثة، الا ان الاسم polygon لن يكون مناسبا! اذن لنسمي هذه الصيغة الاعم للاقتران polyline:

```
def polyline(t, n, length, angle):
   for I in range(n):
      fd(t, length)
      lt(t, angle)
```

الان نعيد كتابة polygon و arc بحيث تَستخدما polyline:

```
def polygon(t, n, length):
   angle = 3600 / n
   polyline(t, n, length, angle)

def arc(t, r, angle):
   arc_length = 2 * math.pi * r * angle / 360
   n = int(arc_length / 3) + 1
   step_length = arc_length / n
   step_angle = float(angle) / n
   polyline(t, n, step_length, step_angle)
```

ثم أخيرا نعدل circle لتستخدم arc:

```
def circle(t, r):
arc(t, r, 360)
```

هذه العملية (اعادة ترتيب البرنامج لتحسين واجمات الاقترانات، و تسهيل اعادة استخدام النص البرمجي) تدعى التفتيت و البناء refactoring. هنا لاحظنا تشابه في النص البرمجي لرسم قوس و ذلك لرسم المضلع، ففتتّنا نص المضلع و أعدنا بناءه ليصبح قوسا.

لو كنا قد خططنا مسبقا لما كان علينا التفتيت و البناء، كان الافضل لنا كتابة polyline منذ البداية، لكننا في الغالب لا نعلم الغيب، ففي بداية المشاريع لا نستطيع تصور النتيجة مئة بالمئة، و لا ما قد يحدث خلال التطوير، فقط عندما تبدأ بكتابة النص البرمجي ستستوعب المشكلة افضل. احيانا يكون التفتيت و البناء علامة على انك تعلمت شيئا جديدا.

4.8 خطة تطوير

خطة التطوير هي عملية كتابة البرامج. العملية التي استخدمناها في دراسة الحالة السابقة هي "الكبسلة و التعميم" خطوات هذه العملية هي:

1. ابدأ بكتابة برنامج صغير بلا تعريفات اقترانات.
2. بمجرد وقوفه على قدميه، غلفه (كبسله) في اقتران و اعط الاقتران اسما.
3. عمم هذا الاقتران باضافة برمترات مناسبة.
4. كرر الخطوات 1-3 إلى ان يصبح لديك مجموعة من الاقترانات الفعالة، انسخ و الصق النصوص البرمجية لتكتب اقل، و لتقلل علاج الاخطاء.
5. ابحث عن الفرص لتحسين البرنامج بالتفتيت و البناء، فمثلا ان كانت هنالك نصوص برمجية متشابهة في عدة مواضع، قيم الفائدة من تفتيتها ثم بنائها في اقتران أعم.

لهذه العملية مساوئها – سترى بدائل لها لاحقا – الا انها مفيدة ان كنت لا تعلم مسبقا كيف ستقسم برنامجك إلى اقترانات، فهذه الطريقة ستمكنك من التصميم بينما تكتب البرنامج.

4.9 جمل التوثيق docstring

هي محارف تكتب في بداية الاقتران لتشرح الواجهة للقاريء (doc = documentation) ، هاك مثل:

```
def polyline(t, n, length, angle):
   """Draws n line segments with the given length and
   angle (in degrees) between them T is a turtle
   """
   for i in range(n):
      fd(t, length)
      lt(t, angle)
```

محارف جمل التوثيق تكون محصورة بين علامات اقتباس ثلاثية، تدعى هذه الجمل أيضا بمحارف متعددة السطور، لأن الثلاثة علامات اقتباس تسمح بتوزيع المحارف على اكثر من سطر.

الجمل فيها تكون موجزة، إلا أنها تحتوي على معلومات ضرورية لمن يريد استخدام هذا الاقتران، تشرح بايجاز ما الذي يفعله الاقتران (من دون الغوص في التفاصيل عن كيف يفعله) و تبين اثر كل من البرمترات على سلوك الاقتران و ما هو نمط كل برمتر (ان لم يكن بيّناً).

كتابة جمل التوثيق هذه جزء مهم من تصميم الواجهة، فواجهة حسنة التصميم تكون سهلة الشرح، و عليه فإن كنت تعاني من تفسير اقترانك، يعني أن هنالك مجال للتحسينات على الواجهة على الارجح.

4.10 علاج الاخطاء

الواجهة مثل العَقد بين الاقتران و المنادي له، فالمنادي يوافق على دفع برمترات معينة و الاقتران يوافق على القيام بعمل معين.

فمثلا `polyline` يحتاج اربعة قرائن: `t` يجب ان تكون سلحفاة، `n` هو عدد القطع المستقيمة، و عليه فيجب ان تكون عددا صحيحا، و `length` يجب أن تكون عددا موجبا، اما `angle` فيجب ان تكون عددا، و المفهوم ضمنا أنه بالدرجات.

هذه المتطلبات تدعى الشروط المسبقة preconditions، لأنها يجب أن تتحقق true قبل ان يبدأ الاقتران بالتنفيذ. المقابل لها هو الشروط الملحقة، و تكون في نهاية الاقتران، تتضمن الشروط الملحقة التأثير الذي سيفعله الاقتران (كرسم قطع مستقيمة) و أي تأثير جانبي (كتحريك السلحفاة أو اي تغيير في عالمها (World).

تكون الشروط المسبقة من مسئولية المنادي، ان خالف المنادي الشروط (و التي وثّقت بعناية) ثم لم يعمل الاقتران بشكل

صحيح، فإن البقَّة على عاتق المستدعي.

4.11 المعاني

تجلّية instance : عضو في مجموعة، TurtleWorld في هذا الفصل كانت عضو في مجموعة TurtleWorld.

حلقة loop: جزء من النص البرمجي ينفَّذ مرارا.

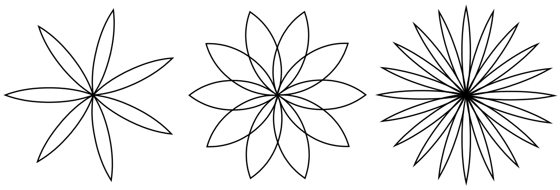
الكبسلة encapsulation: عملية تحويل سلسلة من العبارات إلى تعريف اقتران.

التعميم generalization: عملية تبديل الكائنات التي حُددت من دون ضرورة (كالارقام) بشيء عام مناسب (كالمتغيرات و البرمترات).

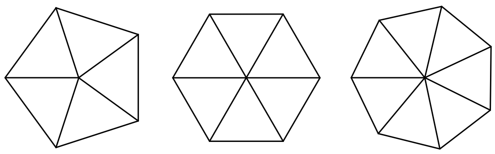
قرينة كلمة مفتاحية keyword argument: قرينة تحتوي اسم البرمتر ك"كلمة مفتاحية".

واجهة interface: وصف لكيفية ستعمال الاقتران، بما فيه اسماء و القرائن و القيم المرجعة و وصفها.

التفتيت و البناء refactoring: عملية تعديل برنامج شغال لتحسين واجهات الاقترانات و لتحسين الخصائص الاخرى للنص البرمجي.



شكل 4.1: زهور السلحفاة



شكل 4.2: فطائر السلحفاة

خطة تطوير development plan: عملية كتابة البرامج.

جمل التوثيق docstring: محارف تظهر في تعريف الاقتران لتوثيق واجهة الاقتران.

الشروط المسبقة precondition: متطلبات يجب الايفاء بها من قبل المنادي قبل ان يبدأ الاقتران.

الشروط الملحقة postconditions: متطلبات يجب أن يفي بها الاقتران قبل ان ينتهي.

4.12 تمارين

تمرين 41 : حمل النص البرمجي لهذا الفصل من

http://thinkpythoncom/code/polygonpy

1. اكتب جملا توثيقية لكل من polygon, arc, circle
2. ارسم رسما مستفا يوضح حالة البرنامج خلال تنفيذه لـ circle(bob, radius) يمكنك القيام بالعمليات الحسابية يدويا ثم اضافة print إلى النص البرمجي.
3. نسخة arc في القسم 47 ليست دقيقة لأن التمثيل الخطي لدائرة يكون دائما خارج الدائرة الحقيقية و لهذا السبب فإن السلحفاة تنتهي بعد المكان المقصود بعدة مديولاتالحل الذي قمت به يوضح طريقة للتقليل من هذا الخطأاقرأ النص البرمجي و انظر ان كنت تستوعبه ان رسمت رسما بيانيا قد يمكنك ان ترى كيف يعمل

تمرين 4.2 اكتب مجموعة اقترانات عامة مناسبة بحيث ترسم زهورا كما في الشكل 4.1

الحل http://thinkpython.com/code/flower.py و يتطلب أيضا

.http://thinkpython.com/code/polygon.py

تمرين 4.3 اكتب مجموعة اقترانات عامة مناسبة بحيث ترسم فطائر كما في الشكل 42الحل

http://thinkpython.com/code/pie.py

تمرين 4.4 بالامكان تشكيل حروف الابجدية الانجليزية من مكونات بسيطة، كخطوط مستقيمة و القليل من المنحنيات، صمم خطا يمكن رسمه بأقل عدد من المكونات البسيطة، ثم اكتب اقترانا يرسم حروف الابجدية.

سيكون عليك كتابة اقتران لكل حرف، و اسماؤها واضحة كـ draw_a و draw_b ثم تخزن اقتراناتك في ملف اسمه .letters.py

للمساعدة في تفحص النص البرمجي يمكنك تحميل turtle typewriter من:

http://thinkpython.com/code/typewriter.py

الحل: http://thinkpython.com/code/letters.py

و يتطلب أيضا http://thinkpython.com/code/polygon.py

تمرين 4.5 اقرأ عن اللوالب من http://en.wikipedia.org/wiki/spiral. ثم اكتب برنامجا يرسم لولبا ارخميديًا (أو أي لولب اخر).

الحل: http://thinkpython.com/code/spiral.py.

الفصل الخامس

المشروطات و الاجترار

5.1 مؤثر مودولوس الحسابي

يعمل مؤثر مودولوس الحسابي على الاعداد الصحيحة، و يرجع الباقي عند قسمة العامل الاول على الثاني. يمثَّل مؤثر مودولوس في بايثون بإشارة النسبة المئوية % و بناؤه نفس بناء الرموز الحسابية الاخرى:

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> reminder = 7 % 3
>>> print reminder
1
```

اذن فناتج قسمة 7 على 3 هو 2 و الباقي 1.

سيظهر لنا ان مؤثر مودولوس مفيد بشكل عجيب، فيمكنك مثلا فحص اذا ما كان الرقم يقبل القسمة على رقم آخر، إن كان ناتج س % ص يساوي صفر فإن س تقبل القسمة على ص.

يمكنك ايضا عزل اخر خانة او خانات من رقم ما. فمحصلة س % 10 ستكون الرقم الموجود في الخانة اليمنى (في الاساس 10) و كذلك س % 100 ستكون العددين في اخر خانتين.

5.2 تعبيرات بوليان

تعبير بوليان هو تعبير يكون أما صحيحا و إما خطأ true , false ، ما يلي هي امثلة تستخدم المؤثر == الذي يقارن عاملين و يرجع True ان كانا متساويين أو False ان لم يكونا:

```
>>> type(true)
<type 'bool'>
>>> type(false)
<type 'bool'>
```

المؤثر == هو أحد مؤثرات النسبة اما باقي مؤثراتها:

```
x != y          #    x لا تساوي y
x > y           #    x أكبر من y
x < y           #    x أصغر من y
x >= y          #    x أكبر أو تساوي y
x <= y          #    x أصغر أو تساوي y
```

هذه المؤثرات قد تكون مألوفة لديك الا ان مؤثرات بايثون تختلف عن المؤثرات الحسابية، استعمال اشارة يساوي = مفردة بدلا من مزدوجة == خطأ شائع، تذكر بأن = هي اشارة تعيين و أن == هي اشارة نسبة فلا يوجد شيء ك =< أو =>.

5.3 المؤثرات المنطقية

هنالك ثلاثة مؤثرات منطقية: `and, or , not` دلالة هذه المؤثرات (معناها) يشبه معناها في الانجليزية، فمثلا

`x > 0 and x < 10` تكون صحيحة فقط إن كانت x أكبر من صفر **و** أصغر من 10.

`n%2 == 0 or n%3 == 0` تكون صحيحة ان كان أي من الشرطين صحيحا، يعني ان كان الرقم يقبل القسمة على 2 **أو** على 3.

و أخيرا الرمز `not` الذي يعكس التعبير البولياني، فإن `not(x > y)` صحيحة إن كانت x > y **ليست** صحيحة، أي أن x اصغر من أو تساوي y.

إن أردنا الدقة في الحديث، فأن عوامل المؤثرات المنطقية يجب أن تكون تعبيرات بوليان، الا ان بايثون ليس صارما جدا، فأي عدد عدا الصفر يفسر بأنه **true**.

```
>>> 17 and True
True
```

بالامكان الافادة من هذه المرونة، الا أن هناك بعض من دقائقها التي قد تشوش، و عليه فالافضل تجنبها (الا ان كنت تعلم ما تفعل).

5.4 التنفيذ المشروط

لكتابة برامج مفيدة، نحتاج إلى القدرة على فحص شروط ما و منثم تعديل سلوك البرنامج بناءا عليها، **العبارات الشرطية** تعطينا هذه القدرة. ابسط صيغة لهذه العبارات هي:

```
if x > 0:
   print 'x is positve'
```

التعبير البولياني بعد `if` يسمى الشرط condition، إن تحقق فإن العبارة المقصودة ستُنفذ، و إن لم يتحقق فلا يحدث شيء.

عبارة `if` لها نفس بناء تعريف الاقتران: ترويسة يتبعها متن بمسافات بادئة. **عبارات كهذه تسمى عبارات مركبة.**

ليس هناك حد لعدد العبارات في متن `if`، فقط يجب أن يحتوي على عبارة واحدة على الاقل. احيانا يكون من المفيد كون المتن بدون عبارات (عادة كمكان محجوز لكتابة نص برمجي لم تستقر عليه بعد). يمكنك في هذه الحالة استخدام عبارة `pass` التي لا تفعل شيء:

```
if x < 0:
   pass         #need to handle negative values!
```

5.5 التنفيذ البديل

شكل آخر لعبارة `if` هو التنفيذ البديل، حيث تكون هناك امكانيتان، و تحقُّق الشرط هو ما يقرر أيهما تنفذ، بناؤها يكون هكذا:

```
if x%2 == 0:
   print 'x is even'
else:
   print 'x is odd'
```

ان كان باقي قسمة x على 2 صفرا، سنعرف بأن x عدد زوجي، فيُظهر لنا البرنامج الرسالة المناسبة، اما ان كانت نتيجة فحص الشرط false، أي لم يتحقق، فإن القسم الثاني من العبارات هو ما سينفذ. بما أن الشرط لا يتعدى أن يكون true أو false فإن بديلا واحدا فقط سينفذ، تسمى البدائل فروع branches لأنها تفريعات في سريان التنفيذ.

5.6 المشروطات المسلسلة

أحيانا تكون هناك أكثر من امكانيتين، و ضرورة وجود أكثر من تفريعتين، احدى طرق التعبير عن عمليات حوسبة كهذه هي المشروطات المسلسلة:

```
if x < y:
   print 'x is less than y'
elif x > y:
   print 'x is greater than y'
else:
   print 'x and y are equal'
```

`elif` هي اختصار `else if`. هنا أيضا ستُنفَّذ تفريعة واحدة فقط. لا يوجد حد لعدد عبارات `elif`، و ان لزم استخدام `else` فيجب وضعها في النهاية، لكن وجود `else` ليس الضروري.

```
If choice == 'a':
   draw_a()
elif choice == 'b':
   draw_b()
elif choice == 'c':
   draw_c()
```

يُفحص كل شرطٍ بالترتيب، فإن كانت نتيجة فحص الشرط الاول false يفحص الشرط الثاني و هكذا. و ما أن تصادف نتيجة فحص true، سينفذ الفرع المعني و تنتهي العبارة، و ان كان هناك أكثر من شرط true فإن أول فرع true فقط سينفذ.

5.7 المشروطات العشية

قد يتطلب الامر أن تكون احدى المشروطات محتضنة (معششة) في أخرى، كان بإمكاننا كتابة الثلاثية السابقة كالتالي:

```
if x == y:
   print 'x and y are equal'
else:
   if x < y:
      print 'x is less than y'
   else:
      print 'x is greater than y'
```

تحتوي المشروطة الخارجية على فرعين، الفرع الاول يحتوي على عبارة بسيطة، الثاني يحتوي على عبارة `if` أخرى، و التي

بدورها تحتوي على فرعين، كل من هذين الفرعين عبارةٌ بسيطة، هذان الفرعان أيضا يمكن جعلهما عبارات مشروطة.

رغم أن المسافات البادئة في هذه الصياغة تجعل قراءة النص سهلة، الا أن المشروطات العشية تصبح بسرعة صعبة القراءة، فتجنُّبها يفضل في الغالب.

المؤثرات المنطقية توفر لنا طريقة لتبسيط المشروطات العشية، فعلى سبيل المثال يمكننا كتابة النص التالي مرة أخرى باستخدام مشروطة واحدة:

```
if 0 < x:
   if x < 10:
      print 'x is a positive single-digit number'
```

عبارة `print` هنا ستنفذ فقط ان جعلناها تمر عبر الشرطين، لذا فبإمكاننا استخدام المؤثر `and` لتحقيق نفس النتيجة:

```
if 0 < x and x < 10:
   print 'x is a positive single-digit number'
```

5.8 الاجترار (العودية) recursion

من الجائز لاقتران نداء اقتران اخر، و من الجائز ايضا لاقتران أن ينادي نفسه، قد لا تكون الفائدة من هذا النداء واضحة الآن، إلا انه سيتبين لنا أن النداء هكذا هو من الامور السحرية التي يستطيع البرنامج القيام بها، فمثلا انظر إلى الاقتران التالي:

```
def countdown(n):
   if n <= 0:
      print 'Blastoff!'
   else:
      countdown(n-1)
```

إن كانت `n` صفرا أو أقل من صفر، ستُطبع الكلمة `'Blastoff!'` و إلا ستُخرِج `n` ثم تنادي اقترانا اسمه `countdown` – نفسه – بتمرير `n-1` كقرينة.

ما الذي سيحدث عندما ننادي هذا الاقتران هكذا؟

```
>>> countdown(3)
```

يبدأ تنفيذ `countdown` بـ `n=3` و بما أن `n` أكبر من 0 ستُخرج القيمة 3، ثم ينادي الاقتران نفسه

سيبدأ تنفيذ `countdown` بـ `n=2` و بما أن `n` أكبر من 0 ستُخرج القيمة 2 ثم ينادي نفسه

سيبدأ تنفيذ `countdown` بـ `n=1` و بما أن `n` أكبر من 0 ستُخرج القيمة 1 ثم ينادي نفسه

سيبدأ تنفيذ `countdown` بـ `n=0` و بما أن `n` ليست أكبر من 0 ستطبع الكلمة `Blastoff!`

ثم يرجع

الاقتران الذي تلقى `n = 1` يرجع

الاقتران الذي تلقى `n = 2` سيرجع

الاقتران الذي تلقى `n = 3` سيرجع

و عندها سنعود إلى `__main__`، فالمخرج الكلي سيكون على هذا الشكل:

```
3
2
1
Blastoff!
```

الاقتران الذي ينادي نفسه يدعى إجتراريا (عوديا) recursive ، العملية نفسها تسمى اجترارا recursion.

و كمثال اخر ، يمكننا كتابة الاقتران الذي يطبع محارف عدد n من المرات:

```
Def print_n(s, n):
   if n <= 0:
      return
   print s
   print_n(s, n-1)
```

إن تحقق n <= 0 فإن عبارة return ستخرجنا من الاقتران، و سريان التنفيذ سيعود مباشرة إلى المنادي، و ما تبقى من سطور الاقتران لن ينفذ.

باقي الاقتران يشبه countdown : إن كانت n أكبر من صفر فستطبع s ثم ينادي نفسه ليطبع s عدد من المرات الاضافية تساوي n - 1. فيكون عدد سطور المخرجات 1 + (n - 1) و تجميعها يكون n.

لأمثلة بسيطة كهذه من الاسهل استخدام حلقة for ، الا أننا سنرى لاحقا أمثلة من الصعب كتابتها بحلقة for و الاسهل كتابتها بالاجترار لهذا استحسنت الابتداء مبكرا.

5.9 الرسم المستف للاقترانات الاجترارية

في القسم 3.10 استعملنا الرسم المستف لتمثيل حالة البرنامج عند نداء اقتران، بإمكان نفس نوع الرسم المساعدة في تفسير الاقتران الاجتراري.

في كل مرة يستدعى فيها الاقتران يخلق بايثون اطار اقتران جديد، يحتوي على متغيرات الاقتران الموضعية و برمتراته في الاقترانات العودية قد يكون هنالك اكثر من اطار في التستيفة في نفس الوقت.

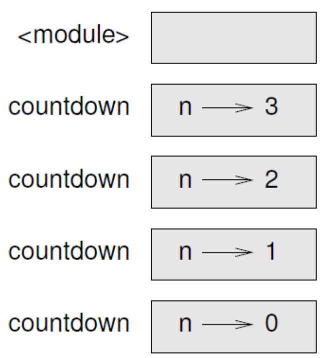
الشكل 5.1 يبين الرسم المستف لـ countdown عند استدعائها بـ n = 1.

كالعادة أعلى الرسم هو اطار لـ __main__ و هو فارغ لأننا لم نوجِد في __main__ أية متغيرات و لم نمرر أية قرائن.

اطارات countdown الاربعة بها قيم مختلفة للبرمتر n ، أسفل الرسم يسمى حالة القاعدة base case و لا يقوم بنداء عودي لذلك فلا يوجد اطارات اضافية.

تمرين 5.1 أرسم رسما تستيفيا لـ print_n نودي بـ 'Hello' = s و n = 2.

تمرين 5.2 اكتب اقترانا و سمه do_n يأخذ كائن اقتران و عدد n كقرائن بحيث يستدعي الاقتران المعطى عدد n من المرات.



شكل 5.1 الرسم المستف

5.10 الاجترار اللا منتهي

إن لم يصل الاجترار إلى الحالة الأساس فسيقوم بنداءات اجترارية إلى الابد، و لن يتوقف تنفيذ البرنامج. يعرف هذا بالاجترار اللا منتهي، و هو في العموم ليس بالامر الجيد، هاك برنامج صغير و اجترار لا منتهي:

```
def recurse():
   recurse()
```

أغلب بيئات البرمجة لا تسمح لبرامج بها اجترار لا منتهي من التنفيذ إلى الابد، و بايثون يصدر رسالة وجود خطأ ان تجاوز عدد النداءات العمق الاقصى:

```
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
            .
            .
            .
File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursions depth exceeded
```

هذه الملاحقة كانت أكبر قليلا مما شاهدنا في الفصل السابق فعند ورود الخطأ يكون عدد الاجترارات قد وصل إلى ألف في التستيفة!

5.11 مدخلات لوحة المفاتيح

كانت البرامج التي كتبناها حتى الان فظة، من حيث أنها لا تقبل مدخلات من المستخدم، فهي تفعل نفس الشيء كل مرة و حسب.

بايثون2 يوفر اقترانا جاهزا اسمه `raw_input` يقبل المدخلات من لوحة المفاتيح، أما في بايثون3 فاسمه `input`. عند نداء هذا الاقتران يتوقف البرنامج و ينتظر بأن يقوم المستخدم بطباعة شيء ما، و عند ضغط المستخدم زر **Enter** أو **Return** فإن البرنامج يتابع، و يرجع الاقتران `raw_input` ما طبعه المستخدم

```
>>> text = raw_input()
What are you waiting for?
>>> print text
What are you waiting for?
```

من المستحسن قبل أخذ المُدخل من المستخدم أن نطبع شيئا يقول للمستخدم ماذا يدخل. هنا `raw_input` تستطيع أخذ محث كقرينة:

```
>>> name = raw_input('What...is your name?\n')
What is your name?
Atrhur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

التسلسل `\n` في نهاية المحث يعني سطرا جديدا، و هو حرف خاص يسبب قطع في السطر، و لهذا يظهر مُدخَل المستخدم في السطر الذي يلي المحث.

ان كنت تتوقع من المستخدم ان يدخل عددا صحيحا فيمكنك محاولة تحويل القيمة المرجعة إلى `int`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
```

```
>>> int(speed)
17
```

اما ان طبع المستخدم ما هو غير محارف من الخانات العددية فستحصل على خطأ:

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European Swallow?
>>> int(speed)
ValueError invalid literal for int() with base 10
```

5.12 علاج الاخطاء

الملاحقة التي يظهرها بايثون عند حدوث الخطأ تحوي من المعلومات الكثير ، الا أنها قد تكون مفرطة أيضا، خصوصا إن كانت هناك اطر عديدة في التستيفة، أكثر اجزاء الملاحقة إفادة هي:

- ما هو نوع الخطأ، و
- أين وقع.

هناك بضعة مآخذ على قولي سابقا بأن الأخطاء الكتابية سهلة الاكتشاف في العادة. فأخطاء الفراغات قد تصبح مخادعة لأن الفراغات و المسافات البادئة لا تُرى و نحن تعودنا على تجاهلها:

```
>>> x = 5
>>>  y = 6
   File "<stdin>", line 1
      y = 6
IndentationError: unexpected indent
```

كانت المشكلة في هذا المثال أن السطر الثاني أبعد بفراغ واحد، الا أن رسالة وجود الخطأ تشير إلى y مما قد يموه الحقيقة بشكل عام تؤشر رسائل وجود الأخطاء المكان الذي لوحظ فيه وجود خطأ الا أن الخطأ الحقيقي قد يكون في سطر أبكر من سطور النص، أحيانا في السطر السابق

يطبق ما سبق أيضا على الأخطاءالتي تظهر عند التشغيل

لنفرض أنك تريد حوسبة نسبة الاشارة – التشويش بالدسبل المعادلة هي $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$.

يمكنك في بايثون كتابة شيء كهذا:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
print decibels
```

الا انك عند تشغيل البرنامج في بايثون2 ستحصل على خطأ.

```
Traceback (most recent call last):
   File "snrpy", line 5, in ?
      Decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

تشير الرسالة إلى وجود خطأ في السطر الخامس، لكننا نرى بأن السطر على ما يرام، لكي نكتشف الخطأ الحقيقي، قد يفيدنا طباعة قيمة `ratio` و سيتضح لنا بأنها 0، فالمشكلة إذن في السطر الرابع، لأن تقسيم عددين صحيحين كان قسمة أرضية،

الحل هنا هو تمثيل قيم `signal_power` و `noise_power` بقيم عائمة، إذن فرسالة وجود الخطأ ستبلغك بمكان اكتشاف الخطأ، و هو ليس المكان الذي سبب الخطأ بالضرورة.

لن تحصل على هذا الخطأ في بايثون3 لأن رمز القسمة هناك يقوم بقسمة نقاط عائمة و إن كانت العوامل أعداد صحيحة.

5.13 المعاني

مؤثر موديولس modulus operator: من المؤثرات الحسابية تمثله اشارة % عوامله أعداد صحيحة و ناتج العملية يكون باقي قسمة العامل على الاخر.

تعبير بوليان Boolean expression : تعبير تكون قيمته إما True أو False.

مؤثر نسبة relational operator: أحد مؤثرات المقارنة بين العوامل: ==, =! , <, > , =< و =>.

مؤثر منطقي logical operator: أحد المؤثرات التي تجمع تعبيرات بوليان: and, or, not.

عبارة مشروطة conditional statement: عبارة تتحكم في سريان التنفيذ بناءا على شرط ما.

شرط condition: تعبير بوليان في عبارة مشروطة يحدد أي من الفروع سيتم تنفيذه.

عبارة مركبة compound statement: عبارة تتألف من ترويسة و متن، تنتهي الترويسة بـ : ، و سطور المتن لها مسافات بادئة بالنسبة للترويسة.

فرع branch: أحد تسلسلات العبارات البديلة في العبارة المشروطة.

المشروطات المسلسلة chained conditional:عبارة مشروطة بها سلسلة من الفروع البديلة

المشروطات العشية nested conditional: عبارة مشروطة تظهر في أحد فروع عبارة مشروطة أخرى.

الاجترار recursion: عملية نداء الاقتران الذي ينفذ حاليا، و يسمى أحيانا بالعودية.

الحالة الاولى base case: فرع مشروط في اقتران اجتراري لا يقوم بنداء اجتراري.

الاجترار اللا منتهي infinite recursion: الاجترار الذي ليس له حالة أساس.

5.14 تمارين

تمرين 5.3 تقول نظرية فِرمات الاخيرة بأنه لا توجد أعداد موجبة صحيحة a و b و c بحيث أن

$$a^n + b^n = c^n$$

لأي قيمة لـ n أكبر من 2.

1. اكتب اقترانا اسمه `check_fermat` يأخذ أربعة برمترات `a, b, c, n` ثم يفحص اذا ما كانت نظرية فرمات ستصمد أم لا فإن كانت n أكبر من 2 و كان حقاً أنَّ

$$a^n + b^n = c^n$$

فعلى البرنامج طباعة "`Holy smokes, Fermat was wrong`" و إلا فعلى البرنامج طباعة "`No, that doesn't work`".

2. أكتب اقترانا يطلب من مستخدمه إدخال قيم لـ `a, b, c` و `n` ثم يحولها لأعداد صحيحة و بعدها يستخدم `check_fermat` ليفحص إذا ما كانت تخالف نظرية فرمات.

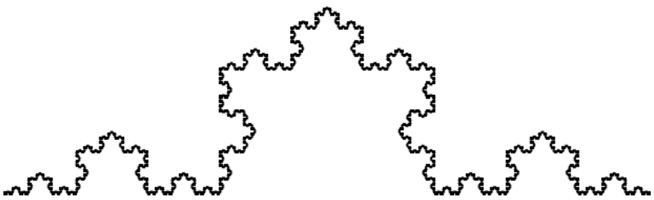
تمرين 5.4 إن أُعطيت ثلاثة عصي، فقد تستطيع أو قد لا تستطيع تكوين مثلث منها، فمثلا إن كان طول إحداها 12 إنشا و طول كل من الاخريين إنشا واحدا يكون جليا أنها لن تكّون مثلثا، فلن تستطيع جعل العصاتين القصيرتين تلتقيان في الوسط. لكن لأي ثلاثة أطوال هنالك طريقة سهلة لمعرفة إن كانت ستكوّن مثلثا أم لا:

إن كان طول أي من الأضلاع أكبر من مجموع طولي الضلعين الآخريين لن تتمكن من تشكيل مثلث، و العكس صحيح (إن كان مجموع طولي الضلعين يساوي الضلع الثالث فإنها تشكل ما اسمه مثلثا متفسخا !).

1. اكتب اقترانا اسمه `is_triangle` يأخذ ثلاثة أعداد صحيحة كقرائن، و يطبع "Yes" أو "No" حسب إمكانية تكوين المثلث من الأطوال المعطاة.
2. أكتب اقترانا يطلب من المستخدم إدخال ثلاثة أطوال للعصي و يحولها لأعداد صحيحة ثم يستخدم `is_triangle` لفحص ما إذا كانت تكون مثلثا

التمارين التالية تستخدم TurtleWorld من الفصل الرابع:

تمرين 5.5 أقرأ الاقتران التالي و حاول معرفة ما الذي يقوم به، ثم شغله (أنظر الامثلة في الفصل الرابع)



الشكل 5.2 منحنى كوخ

```
def draw(t, length, n):
   if n == 0:
      return
   angle = 50
   fd(t, length*n)
   lt(t, angle)
   draw(t, length, n-1)
   rt(t, 2*angle)
   draw(t, length, n-1)
   lt(t, angle)
   bk(t, length*n)
```

التمرين 5.6 منحنى كوخ هو كسيرية fractal كالشكل 5.2 ،لكي ترسم منحنى كهذا كل ما عليك فعله هو:

1. أرسم منحنى كوخ بطول x/3 .
2. در يسارا 60 درجة.
3. ارسم منحنى كوخ بطول x/3 .
4. در يسارا 120 درجة.
5. ارسم منحنى كوخ بطول x/3.
6. در يسارا 60 درجة.
7. أرسم منحنى كوخ بطول x/3.

الاستثناء هو إن كانت x أقل من 3: في هذه الحالة ارسم خطا مستقيما بطول x فقط.

1- اكتب اقترانا و سمه `koch` يأخذ السلحفاة و الطول كبرمترات، ثم يستخدم السلحفاة لرسم منحنى كوخ بالطول المعطى.

2- اكتب اقترانا و سمه `snowflake` ليرسم منحنى كوخ يكون شكله النهائي كبلورة ثلج. الحل: `http://thinkpython.com/code/koch.py`.

3- يمكن تعميم منحنى كوخ بعدة طرق أنظر `http://en.wikipedia.org/wiki/koch_snowflake` و اختر المثال الذي تفضله.

الفصل السادس

الاقترانات المثمرة

6.1 القيمة المرجعة return value

بعض الاقترانات الجاهزة التي استعملناها، كالاقترانات الرياضية، لها نتائج. نداء الاقتران يولد قيمة، نعينها العادة لمتغير أو نضعها في تعبير.

```
A = mathexp(1.0)
height = radius * math.sin(radians)
```

جميع الاقترانات التي كتبناها حتى الان كانت عقيمة، قد تطبع شيئا أو تحرك السلحفاة، الا أن القيمة التي ترجعها كانت دائما None.

في هذا الفصل سنكتب (أخيرا) اقترانات مثمرة. المثال الاول سيكون area و الذي سيرجع مساحة دائرة لها نصف قطر معطى.

```
Def area(radius):
   temp = math.pi * radius**2
   return temp
```

لقد مرت علينا عبارة return من قبل، لكن في الاقترانات المثمرة فإن عبارة return تحتوي على تعبير. هنا هذه العبارة تعني " إرجع فورا من هذا الاقتران و استخدم التعبير التالي كقيمة مرجعة". يمكن للتعبير أن يصبح بالغ التعقيد، و من أجل الإيجاز لكنا قد كتبنا هذا الاقتران هكذا:

```
def area(radius):
   return math.pi * radius**2
```

لكن من الناحية الأخرى، فإن **المتغيرات المؤقتة** مثل temp تسهل اكتشاف الأخطاء.

من المفيد أحيانا ادراج عدة عبارات إرجاع، واحدة في كل فرع في مشروطة:

```
def absolute)value(x):
   if x < 0:
      return -x
   else:
      return x
```

فبما أن عبارات الارجاع هذه موضوعة في مشروطة بدائلية، فإن واحدة منها فقط ستنفذ.

بمجرد تنفيذ عبارة الارجاع، فإن الاقتران سيتوقف بدون تنفيذ أي من العبارات التي تلي عبارة الارجاع return. النص الذي يظهر بعد return أو في أي موضع لا يصله سريان التنفيذ يسمى **نصا برمجيا ميتا**.

في الاقترانات المثمرة، من الجيد أن تتأكد من أن أي مسار برمجي في البرنامج سيصطدم بعبارة إرجاع، مثلا اقتران القيمة المطلقة:

```
def absolute_value(x):
   if x < 0:
      return -x
```

```
   if x > 0:
      return x
```

هنا هذا الاقتران غير صحيح، لأنه إن كانت x صفرا فلن يتحقق أي من الشرطين و سينتهي الاقتران من دون أن يصادف return و إذا وصل سريان التنفيذ إلى نهاية الاقتران ستكون قيمة الارجاع None، و هي ليست القيمة المطلقة لـ **0**.

```
>>> print absolute_value(0)
None
```

بالمناسبة، بايثون يوفر اقترانا جاهزا يدعى abs يحسب القيمة المطلقة.

تمرين 6.1 اكتب اقتران اسمه compare يرجع 1 عندما تكون x > y ، و صفر إن كانت x == y ، ثم 1- إن كانت x < y.

6.2 التطوير العصامي

بينما تتضخم الاقترانات التي تطورها، ستجد نفسك تنفق المزيد من الوقت في علاج الاخطاء.

لتتمكن من التعامل مع البرامج التي تتعقد باضطراد، قد تستفيد من عملية تدعى التطوير العصامي incremental development ، الهدف هو تجنب جلسات علاج الأخطاء الطويلة عن طريق إضافة نصوص برمجية صغيرة مرة بعد مرة.

كمثال، افترض أنك تريد إيجاد المسافة بين نقطتين، المعطيات هي إحداثيات النقطتين (x_1,y_1) و (x_2,y_2)، نظرية فيثاغوروس تقول بأن المسافة هي:

$$\text{المسافة} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

الخطوة الاولى أن تضع تصورا لما سيبدو عليه اقتران distance في بايثون، أي ما هي المعطيات (البرمترات) و ما هي النتائج (القيمة المرجعة return)؟

في حالتنا هذه كانت المعطيات نقطتان، و تُمَثّل في أربعة أرقام، و القيمة المرجعة كانت المسافة و التي تُمَثّل في قيمة حقيقية floating-point.

أنت الان جاهز لكتابة الخطوط العريضة للاقتران:

```
def distance(x1, y1, x2, y2):
   return 0.0
```

من الواضح أن هذه النسخة من برنامجك لن تحسب لنا المسافة، فالقيمة المرجعة ستكون دائمًا صفر، الا أنها صيغة نحوية صحيحة و تعمل، الذي يعني أنه يمكنك اختبارها الآن و قبل أن يزداد تعقيدها.

لفحص الاقتران الجديد، استدعِها بقرينة بسيط:

```
>>> distance(1, 2 ,4 ,6)
0.0
```

اخترت هذه القيم بحيث تكون المسافة الافقية 3 و العمودية 4 و هكذا ستكون النتيجة 5 (وتر المثلث 3-4-5) من الفيد معرفة النتيجة مسبقا عند فحص الاقترانات.

في هذه المرحلة تأكدنا من أن بناء الاقتران صحيح، بإمكاننا الان إضافة النصوص البرمجية لمتن الاقتران. منطقيا، الخطوة التالية ستكون إيجاد الفرق $x_2 - x_1$ و كذلك $y_2 - y_1$ ، إذن فالنسخة التالية من الاقتران ستخزن هذه القيم في متغيرات مؤقتة و ستطبعها:

```
def distance(x1, y1, x2, y2):
   dx = x2 - x1
```

```
    dy = y2 - y1
    print 'dx is', dx
    print 'dy is', dy
    return 0.0
```

إنْ عمِل الاقتران بعد هذه الاضافة فيجب أن يظهر على الشاشة 'dx is 3' و 'dy is 4'. إن حدث هذا فسنعلم بأن الاقتران أخذ القرائن الصحيحة و قام بالحوسبة الاولى بشكل صحيح، و إلا فكل ما بين يدينا هو فقط بضعة سطور لعلاج اخطائها.

التالي هو حساب جمع مربعات dx و dy :

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print 'dx is', dx
    print 'dy is', dy
    dsquared = dx**2 + dy**2
    return 0.0
```

مرة أخرى، شغل البرنامج و تأكد من المخرجات (تعلم مسبقا أنها 25). و أخيرا ستستخدم math.sqrt لإرجاع النتيجة:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print 'dx is', dx
    print 'dy is', dy
    dsquared = dx**2 + dy**2
    result = maths.qrt(dsquared)
    return result
```

إن كانت النتيجة صحيحة فقد انتهيت من كتابة برنامجك، و إن لم تكن، فقد يتطلب الامر طباعة قيمة result قبل عبارة return.

لا تُظهر النسخة الاخيرة من الاقتران أي شيء عند تشغيلها، ترجع قيمة فقط. عبارات print التي كتبناها مفيدة في علاج الاخطاء، لذلك فبمجرد الانتهاء من النص و التأكد من خلوه من الأخطاء فالافضل إزالتها. نص كهذا يدعى السقّالات scuffolding لأنه يساعد في بناء البرنامج لكنه ليس البناء بحد ذاته.

عند البدء بكتابة برنامج عليك أن تضيف سطر أو اثنان في كل مرة، و كلما زادت خبرتك ستجد أنه بإمكانك كتابة و علاج أخطاء نصوص أكبر، و أيا كانت الحال، فإن التطوير العصامي يقلل كثيرا من وقت علاج الاخطاء.

العناصر الرئيسية في هذه العملية هي:

1. ابدأ ببرنامج يعمل، قم بتعديلات صغيرة وعند وجود خطأ في أي مرحلة ستكون لديك فكرة واضحة عن موضعه.
2. استخدم المتغيرات المؤقتة و عين لها القيم التي تريد فحصها.
3. عندما تشعر بالرضى عن عمل البرنامج، تستطيع حذف بعضا من السقّالات. و تستطيع دمج عدة عبارات في تعبير مركب، هذا فقط ان كان الدمج لا يصعّب قراءة البرنامج.

تمرين 6.2 استخدم التطوير العصامي لكتابة اقتران و سمه hypotenuse بحيث يرجع طول وتر مثلث قائم الزاوية إن أعطي طولي الضلعين الاخريين. سجل كل خطوة في العملية.

6.3 التركيب

كما توقعت، فإنه يمكنك نداء اقتران من داخل اخر. هذه الإمكانية تدعى التركيب (أو التوليف composition).

كمثال لها سنكتب اقتران يأخذ نقطتين،إحداها مركز دائرة و الاخرى نقطة على محيطها ثم سيحسب مساحة الدائرة.

لنفرض أننا عينا احداثيات المركز للمتغيرين xc و yc و احداثيات نقطة المحيط xp و yp (المحيط = perimeter) ستكون الخطوة الاولى إيجاد نصف قطر الدائرة، و هو مسافة بين نقطتين، و بما أننا كنا قد كتبنا منذ لحظات اقتران يجد هذه المسافة و سميناه distance:

```
radius = distance(xc, yc, xp, yp)
```

إذن سننتقل مباشرة إلى الخطوة التالية و هي حساب مساحة الدائرة ، لكننا كتبنا هذا الاقتران من قبل أيضا:

```
result = area(radius)
```

كبسلة هاتين الخطوتين في اقتران ستنتج لنا:

```
def circle_area(xc, yc, xp, yp):
   radius = distance(xc, yc, xp, yp)
   result = area(radius)
   return result
```

المتغيرات المؤقتة radius و result مفيدة في التطوير و في علاج الاخطاء، لكن بمجرد عمل البرنامج بشكل صحيح يمكننا الايجاز بدمج عبارات النداء:

```
def circle_area(xc, yc, xp, yp):
   return area(disance(xc, yc, xp, yp))
```

6.4 اقترانات بوليان

تستطيع الاقترانات ارجاع بوليان (True, False)، و هي وسيلة جيدة لإخفاء الفحوص المعقدة داخل الاقتران، مثلا:

```
def is_divisable(x, y):
   if x % y == 0:
      return True
   else:
      return Fals
```

من الشائع إعطاء اقترانات بولين أسماء كأسئلة تجاب بنعم أو لا، is_divisable (هل يقبل القسمة) ستعيد إما صح (True) أو خطأ (False) للتبيين إن كانت x تقبل القسمة على y.

هاك مثال:

```
>>> is_divisable(6, 4)
False
>>> is_divisable(6, 3)
True
```

نتيجة عملية المؤثر == هي بوليان، لذا يمكننا إيجاز الاقتران بإرجاع النتيجة مباشرة:

```
def is_divisable(x, y):
   return x % y == 0
```

اقترانات بوليان تستخدم كثيرا في العبارات المشروطة:

```
if is_divisabel(x, y):
   print 'x is divisable by y'
```

قد يغرينا هذا الامر بكتابة شيء كهذا:

```
if is_divisable(x, y) == True:
   print 'x is divisable by y'
```

الا أن المقارنة الثانية كانت زائدة و غير ضرورية.

تمرين 6.3 اكتب الاقتران `in_between(x, y, z)` الذي يرجع `True` إن كانت $x \le y \le z$ و إلا سيرجع `False`.

6.5 المزيد من الاجترار

لقد غطينا جزءا يسيرا من بايثون فقط، الا أنك ستعجب إن علمت بأن هذا الجزء هو لغة برمجة **كاملة**، مما يعني أن أي شيء يمكن حوسبته، يمكن التعبير عنه بهذه اللغة. أي برنامج استخدمته يمكنك كتابة مثيل له باستخدام المزايا التي تعلمتها حتى الان فقط (للدقة، ستحتاج لبضعة أوامر للتحكم بأدوات كلوحة المفاتيح، الفأرة، الاقراص إلا أن هذا كل ما هنالك).

للبرهان على صحة هذا الادعاء، هنالك تمرين غير عادي صممه لأول مرة واحد من أوائل علماء الحاسوب ألن تورنغ، البعض يحاجج بأنه كان رياضيا، الا أن كثير من علماء الحاسوب الاوائل كانوا رياضيين) سمي هذا التمرين أطروحة تورنغ. للإطلاع على نقاش الاطروحة بشكل موسع (و دقيق) أوصي بكتاب ميشيل سبسر "مدخل إلى نظرية الحوسبة".

لكي تأخذ فكرة عما يمكنك عمله بالأدوات التي تعلمت استخدامها حتى الان، سنقوم بتقييم بضعة اقترانات رياضية إجترارية التعريف. التعريف الاجتراري لاقتران يشبه التعريف التدويري، من منطلق أن التعريف يتضمن إشارات للمعرِّف، التعريف التدويري بحد ذاته عديم الفائدة في العموم:

فوربال: هي صفة لما هو فوربال.

إن رأيت ما سبق في معجم ستنزعج، في المقابل إن بحثت عن تعريف اقتران المضروب، و الذي يرمز له بـ !، ستجد شيئا كهذا:

```
0! = 1
n! = n(n - 1)!
```

يقول لنا هذا التعريف بأن مضروب 0 هو 1 ، و مضروب أي قيمة أخرى n هو nمضروبا بمضروب n-1

جسنا، فمضروب 3 هو 3 ضرب !2، و الذي هو 2 ضرب !1، و الذي هو 1 ضرب !0 عندما تقوم بكل العمليات الحسابية فإن !3 هي 3 ضرب 2 ضرب 1 ضرب 1، و بالتالي 6.

إن كان باستطاعتك كتابة تعريف اجتراري لشيء ما، فعلى الاغلب سيكون بوسعك كتابة برنامج بايثون يقيّمه. الخطوة الاولى هي أن تقرر ماذا ستكون البرمترات، في حالتنا هذه يجب أن يكون واضحا لنا بأن `factorial` تأخذ أعدادا صحيحة:

```
def factorial(n):
```

و إن كان هذا العدد الصحيح صفرا فعلى الاقتران أن يرجع 1:

```
def factorial(n):
   if n == 0:
      return 1
```

و إلا، و هذا هو الجزء المهم، علينا القيام بنداء اجتراري لإيجاد مضروب n-1 ثم نضربه بـ n:

```
def factorial(n):
```

```
if n == 0:
    return 1
else:
    recurse = factorial(n-1)
    result = n * recurse
    return result
```

سريان التنفيذ في هذا البرنامج يشبه ذلك في countdown في القسم 5.8 فإن استدعينا factorial بالقيمة 3 :

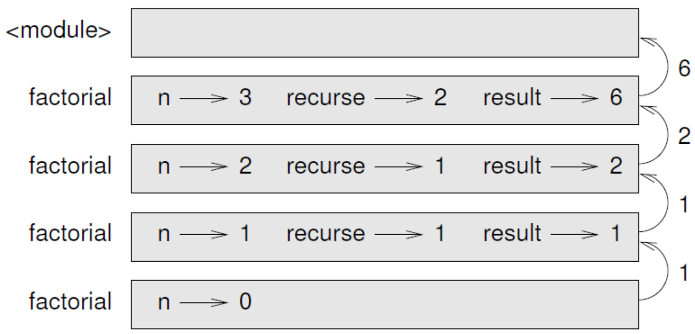
بما أن 3 ليست صفرا، سنمر في الفرع الثاني و نحسب قيمة مضروب n-1

بما أن 2 ليست صفرا سنمر في الفرع الثاني و نحسب قيمة مضروب n-1
و بما أن 1 ليست صفرا سنمر في الفرع الثاني و نحسب قيمة مضروب n-1
بما أن 0 هي 0، سنأخذ الفرع الاول و نعيد 1 دون القيام بنداءات اجترارية أخرى
ستضرب القيمة المرجعة (1) بـ n و هي 1، و سترجَع النتيجة
النتيجة المرتجعة 1 ستضرب بـ n و هي 2 و سترجَع النتيجة
القيمة المرتجعة 2 ستضرب بـ n و هي 3 و النتيجة، 6، ستصبح القيمة التي سيرجعها نداء الاقتران الذي بدأ هذه العملية.

الشكل 6.1 يبين الرسم المستف لهذا التسلسل في نداء اقتران

تظهر القيم المرجعة و هي تمرر عائدة إلى الاعلى في التستيفة ففي كل اطار تكون القيمة المرجعة هي قيمة return و التي هي حاصل ضرب n في recurse.

في الاطار الاخير تختفي المتغيرات الموضعية، لأن الفرع الذي أوجدها لم ينفذ.



الشكل 6.1 رسم مستف

6.6 وثبة ثقة

تتبع سريان التنفيذ هو احدى طرق قراءة البرامج، الا أنه سرعان ما يصبح البرنامج كمتاهة. البديل لهذه الطريقة هي ما يسمى قفزة الثقة. عندما تصل إلى عبارة نداء اقتران، بدلا من اتباع سريان التنفيذ، افترض بأن الاقتران يعمل بشكل صحيح و يرجع النتيجة الصحيحة.

هذا ليس بجديد، فأنت تقوم بقفزات الثقة هذه عندما تستخدم الاقترانات الجاهزة، عندما تستدعي math.cos أو

math.exp فأنت لا تتفحص متن هذه الاقترانات، تفترض فقط بأنها صحيحة لأن من كتب هذه الاقترانات مبرمجون عتاة.

ينطبق نفس الشيء عندما تكتب أقترانك. فمثلا في القسم 6.4 كتبنا اقترانا اسمه is_divisible يقرر اذا ما كان عدد ما يقبل القسمة على اخر، و بمجرد أن أقنعنا أنفسنا بأن الاقتران أصبح صحيحا – بعد فحص النص و تجربة الاقتران – أمكننا استخدام الاقتران دون النظر في متنه ثانية.

نفس الشيء صحيح بالنسبة للبرامج الاجترارية فعندما تصل إلى عبارة نداء اجترارية، بدلا من اتباع سريان التنفيذ، عليك افتراض صحة عمل النداء الاجتراري (و أنه يصدر النتيجة الصحيحة) ثم تسأل نفسك "لو كان بإمكاني ايجاد مضروب n-1 هل بوسعي حساب مضروب n؟" في هذه الحالة، واضح أنه يمكنك، عن طريق الضرب في n.

طبعا قد يكون افتراض أن الاقتران يعمل غريبا بعض الشيء عندما لم تنته من كتابته بعد، الا أن هذا هو سبب تسمية الطريقة بـ قفزة ثقة.

6.7 مثال آخر

في العادة ، بعد المضروب كمثال لشرح الاجترار يأتي فبوناشي، و له هذا التعريف أنظر

:(http://en.wikipedia.org/wiki/Fibonacci_number)

فِبوناشي(0) = 0

فبوناشي(1) = 1

فبوناشي(n) = فبوناشي(n - 1) + فبوناشي(n - 2)

عند كتابته في بايثون سيبدو هكذا:

```
def fibonacci (n):
   if n == 0:
      return 0
   elif n == 1:
      return 1
   else:
      return fibonacci(n−1) + fibonacci(n−2)
```

ان حاولت تتبع سريان التنفيذ هنا، حتي لقيمة صغيرة لـ n سينفجر دماغك، لكن حسب قفزة الثقة، ان افترضت بأن النداءين الاجترارين يعملان بشكل صحيح، سيكون من الواضح ان النتيجة الصحيحة ستأتي عند جمعهما معا.

6.8 فحص الانماط

ما الذي سيحدث عندما نستدعي factorial و نمرر له 1.5 كقرينة

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

يبدو كاجترار لا نهائي، و لكن هل يمكن هذا؟ فهناك حالة أساس – وهي عندما تكون n == 0.

لكن ما يحدث هو أنه عندما لا تكون n عددا صحيحا فقد لا نصادف حالة الاصل و نجتر إلى الابد.

في النداء الاجتراري الاول كانت قيمة n تساوي 0.5 و في النداء الذي تلاه كانت 0.5- و بعدها ستستمر القيمة في النقصان

(سالبا) الا أنها لن تصل إلى الصفر.

هنا يصبح لدينا خيارين، إما تعميم factorial لتعمل على الاعداد الحقيقية، أو جعل factorial يفحص نمط القرينة. الخيار الاول يدعى اقتران جاما و هو أبعد قليلا عن الفكرة من هذا الكتاب، لذلك سنأخذ الخيار الثاني.

بوسعنا استخدام اقتران جاهز يسمى isinstance للتأكد من نمط القرينة و بينما نفعل ذلك بوسعنا أيضا التأكد من أن قيمة القرينة موجبة:

```
def factorial (n):
   if not isinstance(n, int):
      print 'Factorial is only defined for integers'
      return None
   elif n < 0:
      print 'Factorial is not defined for negative integers'
      return None
   elif n == 0:
      return 1
   else:
      return n * factorial(n-1)
```

حالة الأساس الاولى تتعامل مع الاعداد غير الصحيحة، و الثانية تلتقط الاعداد السالبة، في كلتا الحالتين سيطبع البرنامج رسالة وجود خطأ و يرجع قيمة None للإعلان بأن شيئا ما كان خطأ:

```
>>> factorial('fred')
Factorial is only defined for integers
None
>>> factorial(-2)
Factorial is not defined for negative integers
None
```

و ان تمكنا من عبور الفحصين، سنعرف بأن n إما موجبة أو صفر، إذن فبإمكاننا اثبات أن الاقتران سينهي عمله

يمثل هذا البرنامج نمطا يسمى أحيانا الوصي gardian فالمشروطتين الاوليين تعملان كوصي، تحميان النص من قيم قد تسبب خطأ، الوصي يمكّننا من التأكد من صحة النص البرمجي.

في القسم 11.3 سنرى بدائل أكثر مرونة لطباعة رسائل وجود الخطأ: رفع استثناء.

6.9 علاج الاخطاء

تقسيم نص برمجي طويل إلى أجزاء يخلق محاسيم طبيعية تفيد في علاج الأخطاء. ان كان الاقتران لا يعمل، فهناك ثلاثة احتمالات:

- هناك خطأ في القرائن التي مررت للاقتران، مخالفة لأحد الشروط المسبقة.
- هناك خطأ في الاقتران، مخالفة لأحد الشروط الملحقة.
- خطأ في القيمة المرجعة أو في طريقة استعمالها.

لكي تستثني الاحتمال الاول يمكنك اضافة print في بداية الاقتران لتطبع قيم البرمترات (و ربما أنماطها) أو حتى أنه يمكنك كتابة نص مخصص لفحص الشروط المسبقة

ان كانت البرمترات على ما يرام، اضف print قبل كل عبارة return تطبع قيمة الارجاع و إن أمكنك فتأكد من

النتيجة يدويا، قد يكون من المفيد نداء الاقتران بقيم سهلة ليتسنى فحص النتائج بسرعة (كما في القسم 6.2)

ان بدا أن الاقتران يعمل بشكل صحيح، فانظر في نداء الاقتران و تأكد من أن القيمة المرجعة تستعمل بشكل صحيح (أو إن كانت تسعمل أصلا).

اضافة عبارة `print` في بداية و نهاية الاقتران يساعد في جعل سريان التنفيذ مرئيا فمثلا، هنا نسخة من `factorial` بها عبارات `print`:

```
def factorial(n):
   space = ' ' * (4 * n)
   print space, 'factorial', n
   if n == 0:
      print space, 'returning 1'
      return 1
   else:
      recurse = factorial(n-1)
      result = n * recurse
      print space, 'returning', result
      return result
```

هنا `space` هي محارف من الفراغات، تتحكم بالمسافات البادئة للمخرجات. ها هي نتيجة `factorial(5)`:

```
               factorial 5
            factorial 4
         factorial 3
      factorial 2
   factorial 1
factorial 0
returning 1
   returning 1
      returning 2
         returning 6
            returning 24
               returning 120
```

هذه النتيجة توضح معنى سريان التنفيذ، و يمكن لمخرج كهذا أن يكون مفيدا تركيب سقالات فعالة قد يأخذ بعض الوقت، الا أنه سيوفر وقتا أطول عند علاج الاخطاء

6.10 المعاني

متغير مؤقت temporary variable: متغير يستخدم للإحتفاظ بالقيم مؤقتا في عملية حسابية معقدة.

نص ميت dead code: جزء من البرنامج لن ينفذ أبدا، غالبا لأنه يظهر بعد عبارة `return`.

`None`: قيمة خاصة يرجعها اقتران ليس به عبارة `return` أو أن عبارة الارجاع بدون قرائن.

التطوير العصامي incremental development: خطة تطوير برنامج، هدفها التقليل من علاج أخطائه عن طريق اضافة و فحص القليل من النص البرمجي في كل مرة.

السقالات scuffolding: نص برمجي يستخدم خلال تطوير البرنامج لكنه ليس جزءا من المنتج النهائي.

الوصي Guardian: نمط برمجي يستعمل العبارات المشروطة للفحص و التعامل مع الظروف التي قد تولد خطأ.

6.11 تمارين

تمرين 6.4 ارسم رسما مستفا للبرنامج التالي ما الذي سيطبعه البرنامج؟

الحل: `http://thinkpython.com/code/stack_diagram.py`.

```
def b(z):
   prod = a(z, z)
   print z, prod
   return prod

def a(x, y):
   x = x + 1
   return x * y

def c(x, y, z):
   total = x + y + z
   square = b(total)**2
   return square

x = 1
y = x + 1
print c(x, y+3, x+y)
```

تمرين 6.5 اقتران أكرمان، $A(m, n)$ يعرف كالتالي:

$$A(m,n) = \begin{cases} n+1 & if\ m = 0 \\ A(m-1,1) & if\ m > 0\ and\ n = 0 \\ A(m-1, A(m, n-1)) & if\ m > 0\ and\ n > 0 \end{cases}$$

أنظر `http://en.wikipedia.org/wiki/Ackermann_function`.

اكتب اقترانا و سمه `ack` تقيم به اقتران أكرمان، استخدم اقترانك لتقييم `ack(3, 4)` و الذي يجب أن يكون 125. ماذا سيحدث عند تعيين قيم أكبر لـ `m` و `n` ؟

الحل: `http://thinkpython.com/code/ackermann.py`.

تمرين 6.6 الكلمات المتناظرة palendromes هي كلمات تقرأ من اليسار إلى اليمين كما تقرأ من اليمين إلى اليسار، مثل سوس و سلس و noon و redivider. اجترازيا، تكون الكلمة متناظرة إن كان أولها و اخرها نفس الحرف ثم كان ما تبقى كلمة متناظرة.

الاقتران التالي يأخذ محارف كقرينة و يرجع الحرفين الاول و الاخير و حروف الوسط:

```
def first(word):
   return word[0]

def last(word):
   return word[-1]
```

```
def middle(word):
   return word[1:-1]
```

سنرى طريقة عمل هذه التعريفات في الفصل الثامن.

1. اطبع هذه الاقترانات الثلاث في في ملف و سمه palindrome.py ثم تفحص عملها ما الذي سيحدث ان استدعيت middle بسلسلة من حرفين؟ حرف واحد؟ و ماذا عن محارف فارغة، و التي تكتب هكذا '' ؟
2. اكتب اقترانا اسمه is_palindrome يأخذ محارف كقرينة و يرجع True ان كانت متناظرة و إلا False تذكر أنه يمكنك استخدام الاقتران الجاهز len لمعرفة عدد الحروف.
الحل: http://thinkpython.com/code/palindrome_soln.py.

تمرين 6.7 العدد a هو أس العدد b اكتب اقترانا اسمه is_power يأخذ برمترين a و b و يرجع True إن كان a هو أس b، تذكر بأن عليك التفكير بحالة الاصل.

تمرين 6.8 القاسم المشترك الاعظم لـ a و b هو أكبر عدد يقسم الرقمين عليه دون باقٍ،احدى طرق إيجاد **ق-م-أ** لرقمين هو ايجاد باقي قسمة a على b ثم يكون $gcd(a, b) = gcd(b, r)$ ، كحالة أصل يمكننا استخدام $gcd(a, 0) = a$.

اكتب اقترانا اسمه gcd يأخذ برمترين a و b و يرجع قاسمهما المشترك الاعظم.

عرفان: بني هذا التمرين على مثال من "هيكل و تأويل البرامج الحاسوبية" لـ أبلسون و سُسْمن.

الفصل السابع

التكرار

7.1 تعدد التعيينات

قد تكون قد اكتشفت لوحدك جواز وجود أكثر من تعيين لمتغير واحد، التعيين الجديد يجعل المتغير يشير إلى قيمة جديدة (و يتوقف عن الاشارة إلى السابقة).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

مخرجات هذا البرنامج هي 5 7 ، لأنه عند طباعة bruce الاولى كانت قيمته 5، و في الثانية كانت 7 الفاصلة في نهاية سطر print الاولى تمنع الانتقال لسطر جديد، لذلك ظهرت المخرجات على نفس السطر.

الشكل 7.1 يبين كيف يبدو تعدد التعيينات في رسم الحالة.

من المهم عند تعدد التعيين التفريق بين عملية تعيين و عملية مساواة، و لأن بايثون يستخدم اشارة التساوي (=) للتعيينات، فقد نميل إلى تفسير عبارة كـ a = b على أنها عبارة مساواة و هذا غير صحيح.

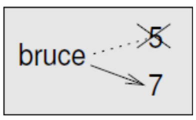
بادئا ذي بدئ، المساواة علاقة تماثلية و التعيين لا، فمثلا، في الرياضيات، إن كانت a = 7 فإن 7 = a في بايثون a = 7 قانونية أما 7 = a فليست.

و أكثر من هذا، في الرياضيات، تكون عبارات المساواة إما صائبة أو خاطئة طوال الوقت في بايثون عبارة التعيين قد تساوي متغيرين، لكن ليس بالضرورة أن يظلّا متساويين.

```
a = 5
b = a # a and b are now equal
a = 3 # a and b are no longer equal
```

السطر الثالث يغير قيمة a و ليس b ، إذن فقد زال تساويهما

رغم الفائدة الكبيرة للعيينات المتعددة، الا أن عليك توخي الحذر عند استعمالها. ان تبدلت قيم المتغيرات كثيرا سيصبح من الصعب قراءة و علاج أخطاء النص البرمجي.



الشكل 7.1: رسم الحالة

7.2 تحديث المتغيرات

احد أشكال التعيين المتعدد هو التحديث، حيث تعتمد القيمة الجديدة لمتغير على سابقتها.

```
x = x +1
```

معنى هذا "خذ القيمة الحالية لـ x ، أضف لها 1 ، ثم حدثها بالقيمة الجديدة".

إن حاولت تحديث متغير غير موجود ستحصل على خطأ، لأن بايثون يقيم الجانب الايمن قبل تعيين القيمة لـ x:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

قبل أن تتمكن من تحديث متغير عليك أن تستهل المتغير أولا، عادة بتعيين بسيط:

```
>>> x = 0
>>> x = x + 1
```

تحديث المتغير بإضافة 1 تدعى increment، أما بطرح 1 فتسمى decrement.

7.3 عبارة while طالما

أكثر ما تُستخدم الحواسيب فيه هو أتمتة المهام المتكررة، فتكرار عمل المهام نفسها و بدون أخطاء هي ما أفضل يقوم به الحاسوب، و ما يفتقر اليه البشر.

لقد رأينا برنامجين، countdown وprint_n، يستخدمان الاجترار لإعادة القيام بالعملية، هذا يدعى أيضا بالتكرار. و لكون التكرار مفهوم شائع، وفّر بايثون عدة مزايا لغوية تسهل استخدامه، إحداها هي عبارة for التي رأيناها في القسم 4.2، سنعود لها لاحقا.

المزية الاخرى هي عبارة while. هذه نسخة من countdown تستخدم عبارة while :

```
def countdown(n):
   while n > 0:
      print n
      n = n-1
   print 'Blastoff!'
```

بإمكانك قراءة عبارة while كما لو كانت انجليزية (كذلك: "طالما" كما لو كانت عربية) و هي تعني "طالما n أكبر من صفر، إطبع قيمة، n ثم أنقصها بمقدار 1، و عندما تصل إلى الصفر أظهر الكلمة !Blastoff"

رسميا أكثر، هذا سيران النتفيذ لعبارة while :

1- قيّم الشرط بإخراج إما True أو False.
2- ان كانت النتيجة False ، إخرُج من عبارة while و أكمل التنفيذ من العبارة التي تليها.
3- ان كانت النتيجة True نفذ ما في متن while ثم عد إلى الخطوة 1.

يدعى هذا النوع من سريان التنفيذ بـ الحلقة Loop لأن الخطوة الثالثة عادت مرة أخرى إلى بداية التنفيذ.

يجب على متن الحلقة أن يغير قيمة متغير أو أكثر لكي ينتهي الشرط إلى False فيُوقف الدوران في الحلقة. و الا فسيكون الدوران في حلقة الى الأبد و عندها تصبح الحلقة **لا منتهية** (حلقة مفرغة). علماء الحاسوب يرون التعبير على علب الشامبو ظريفا "صوبن، اشطف ثم كرر" يذكرهم بالحلقة المفرغة.

في حالة countdown أثبتنا أن الدوران في الحلقة سيتوقف لأن قيمة n منتهية، و لأننا نرى بأن قيمة n آخذة بالصِّغر في كل مرة تُدَّور بها في الحلقة، إلى أن تنتهي إلى الصفر، في حالات اخرى يكون من الصعب الاثبات:

```
def sequence(n):
   while n != 1:
      print n,
      if n%2 == 0: # n is even
         n = n/2
      else: # n is odd
         n = n*3+1
```

الشرط في هذه الحلقة هو أن n!=1، إذن فالدوران سيستمر إلى أن تصبح n تساوي 1 مما يجعل الشرط false.

في كل دورة في الحلقة، يُخرج البرنامج قيمة لـ n ثم يفحص إذا ماكانت فردية أو زوجية، إن كانت زوجية فستُقسَم n على 2 ، و إن كانت فردية فستُستبدل قيمة n بـ n*3+1، مثلا إن كان القرينة التي مُررت إلى sequence هي 3، فالتسلسل المخرج هو 1 ,2 ,4 ,8 ,16 ,5 ,10.

بما أن n تزداد أحيانا و تنقص أخرى، فلا يوجد ما يبرهن بأنها ستصل إلى 1، أو أن البرنامج سيتوقف. لقيم معينة لـ n نستطيع البرهنة، مثلا إن كانت القيمة الابتدائية هي الأس 2، فستكون قيمة n زوجية في كل مرة تدور فيها في الحلقة إلى أن تنتهي إلى 1، سينتهي المثال السابق بهذا المسار كهذا ان بدئ بـ 16.

السؤال الاصعب هو هل يمكننا اثبات أن البرنامج سيتوقف لكل قيم n لم يتمكن أحد حتى الان من اثبات أو نفي هذه الامكانية!

أنظر: http://en.wikipedia.org/wiki/Collatz_conjecture

تمرين 7.1 أعد كتابة الاقتران print_n من القسم 5.8 مستخدما مبدأ التكرار بدلا من مبدأ الاجترار.

7.4 الكبح **break**

أحيانا لا تتمكن من معرفة إذا ما حان الوقت للاقتران أن يتوقف قبل أن تصل منتصف المتن.

في هكذا حالات يمكنك استخدام عبارة break للقفز خارج الحلقة.

فعلى سبيل المثال افترض أنك تريد من البرنامج أن يصِرّ على طلب مدخلات من المستخدم، لكن إن طبع المستخدم done يتوقف عن الاصرار، بإمكانك كتابة:

```
while True:
   line = raw_input('> ')
   if line == 'done':
      break
   print line
print 'Done!'
```

شرط الحلقة هو True، و الذي سيكون متحققا دائما، إذن فهذه الحلقة ستظل تعمل إلى أن تصادف عبارة break.

في كل دورة سيصغي المحث للمستخدم بقوس زاوي. فإن طبع المستخدم done، ستخرجنا عبارة break من الحلقة، و إلا فسيردد البرنامج ما يطبعه المستخدم ثم يعود إلى رأس الحلقة، هذا تشغيل البرنامج:

```
> not done
not done
> done
Done!
```

هذه الطريقة لكتابة حلقة while شائعة لأنها تستطيع فحص الشرط في موضع من الحلقة (و ليس فقط في قمتها). و تمكننا من التعبير بحزم عندما نريد للتنفيذ أن يتوقف (" قف عند حدوث هذا") بدلا من (" استمر حتى يحدث هذا")

7.5 الجذور التربيعية

تستخدم الحلقات كثيرا لحساب النتائج العددية عن طريق البدء بقيمة مقربة للجواب ثم استخدام التكرار لتحسينه.

مثلا، احدى طرق حساب الجذر التربيعي هي طريقة نيوتن. افرض أن لديك عدد a و تريد معرفة جذره التربيعي. إن ابتدأت بأي توقع، x ، فبإمكانك حساب توقع أقرب، حسب المعادلة التالية:

$$y = \frac{x + a/x}{2}$$

لنفرض مثلا أن a كانت 4 و x كانت 3:

```
>>> a = 4.0
>>> b = 3.0
>>> y = (x + a/x)/2
>>> print y
2.166666667
```

هذه النتيجة أقرب من 3 إلى الجواب الصحيح ($\sqrt{4} = 2$). الان، إن استخدمنا هذه النتيجة و دورناها في نفس المعادلة سنقترب أكثر إلى الجواب الصحيح:

```
>>> x = y
>>> y = (x + a/x)/2
>>> print y
2.00641025641
```

بعد بضعة تحديثات سيكون الجواب تقريبا مئة بالمئة:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00000000003
```

نحن لا نعلم مسبقا ما هو عدد الخطوات التي سيتطلبها ايجاد الجواب الصحيح، الا أننا سنعلم بأننا وصلنا إلى الجواب الصحيح لأن سلوك الحلقة سيتغير:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
>>> x = y
```

```
>>> y = (x + a/x) / 2
>>> print y
2.0
```

عندما تصبح `x == y` بإمكاننا التوقف. هذه حلقة تبدأ بتوقع أوَّلي، x، ثم تظل تُحسِّنه إلى أن تتوقف الحلقة عن التغير:

```
while True:
   print x
   y = (x + a/x) / 2
   if y == x:
      break
   x = y
```

هذه الطريقة تعمل جيدا لمعظم قيم a الا أنها خطرة في العموم عند فحص التساوي لـ `float`. قيم النقاط العائمة "صحيحة تقريبا" فقط: فلا يمكن تمثيل الاعداد الدورية كالثلث و اللادورية مثل $\sqrt{2}$ مئة بالمئة عن طريق الاعداد العائمة.

الآمَن من فحص ان كانت كل من x و y متساويتان، هو استخدام الاقتران الجاهز abs الذي يحسب القيمة المطلقة، أو الارتفاع، للفرق بينهما

```
   If (y-x)<epsilon:
      break
```

حيث `epsilon` لـها قيمة ما مثل `0.0000001` التي تحدد ماهو "القريب بما فيه الكفاية".

تمرين 7.2 كبسل هذه الحلقة في اقتران اسمه `square_root` يأخذ a كبرمتر ثم يختار قسمة معقولة لـ x و يرجع توقعا للجذر التربيعي لـ a.

7.6 الخوارزميات Algorithms

كانت طريقة نيوتن مثالا للخوارزميات: عملية رياضية لحل فئة من المسائل (حساب الجذور التربيعية في هذه الحالة).

ليس من السهل تعريف الخوارزمية، لكن قد يقرِّب معناها البدء بشيء ليس خوارزميا. في المدرسة، عندما تعلمت جداول الضرب لخانة واحدة كان عليك تذكر جدول الضرب كاملا، أي أنك حفظت مئة حل لمئة مسألة. هذا النوع من المعرفة ليس خوارزميا.

الا أنه إن كنت "كسولا" حينها لكنت قد غششت بأن تعلمت بضعة خدع، فمثلا لتوجد حاصل ضرب **س** في 9، كنت تكتب **س** - 10 في الخانة الاولى و 10 – **س** في الخانة الثانية، هذه الخدعة هي طريقة عامة لحل حاصل ضرب 9 في الارقام ذات الخانة الواحدة. هذا الاسلوب كان خوارزميا!

كذلك كانت المهارات التي تعلمتها في الجمع بالحمل و الطرح بالاقتراض و القسمة الطويلة، كلها خوارزميات. من صفات الخوارزميات أنها لا تتطلب ذكاءا للقيام بها، هي فقط عمليات ميكانيكية تتبع فيها كل خطوة الخطوة التي سبقتها بناءا على قوانين سهلة. برأيي أنه من السخف للإنسان أن يمضي أوقاتا طويلة في المدارس يتعلم كيف **ينفذ** الخوارزميات التي، حرفيا، لا تتطلب أي ذكاء.

في المقابل فإن عملية تصميم الخوارزميات هي المثيرة، و بها تحد ثقافي، و هي جزء مركزي مما نسميه البرمجة.

بعض الاشياء التي يقوم بها الناس بشكل طبيعي و بلا صعوبة، أو حتى بلا وعي، تكون الاصعب عند محاولة التعبير عنها خوارزميا. مثال جيد لهذا هو فهم اللغات الطبيعية ، كلنا نتكلمها (نفعلها) لكن لم يتمكن أحد حتى الان أن يفسر "كيف" نفعلها، ليس تفسيرا على شكل خوارزمية على الاقل.

7.7 علاج الاخطاء

عندما تبدأ بكتابة برامج أكبر، قد تجد نفسك غارقا لمدة أطول في علاج الأخطاء. نص برمجي أطول يعني إحتمالات أكثر للخطأ و أمكنة أكثر ليختبئ فيها البق.

إحدى طرق تقليل الوقت المستنفذ هي "علاج الأخطاء بالتنصيف"، فإن كان لديك 100 سطر في البرنامج، ستأخذ في العادة 100 خطوة للبحث عن الأخطاء و علاجها.

بدلا من ذلك حاول تقسيم المشكلة إلى نصفين ابحث في منتصف النص أو بالقرب منه عن قيمة متوسطة تستطيع متابعتها أضف عبارة `print` أو أي شيئٍ آخر له أثر تأكيدي، ثم شغل البرنامج.

أن كانت نقطة البحث الوسطى هذه غير صحيحة فلابد أن يكون الخطأ في النصف الاول من النص، و ان كانت صحيحة سيكون في النصف الثاني.

في كل مرة تقوم بفحص كهذا تنَصِّف عدد السطور التي بقي عليك البحث فيها عن الخطأ، بعد 6 خطوات (أقل من المئة خطوة السابقة) سيتبقى لك سطر أو سطرين من النص البرمجي لتفحصه، على الاقل نظريا.

عمليا لا يكون موقع منتصف المشكلة واضحا، و أحيانا غير قابل للفحص، و من غير المعقول أن تعد سطور البرنامج لتصل إلى المنتصف. بدلا من ذلك فكر بالاماكن التي تتوقع وجود خطأ فيها، و الاماكن التي من السهل وضع مقسوم بها، ثم اختر البقعة التي يكون احتمال اختباء البقة قبلها أقرب من احتماله بعدها.

7.8 المعاني

تعدد التعيينات multiple assignment: عمل أكثر من تعيين لمتغير خلال تشغيل البرنامج.

تحديث update: تعيين تعتمد فيه القيمة الجديدة لمتغير على القديمة.

استهلال initialization: تعيين يعطي القيمة الابتدائية لمتغير سيتم تحديثه.

زيادة increment : تحديث تزاد فيه قيمة المتغير (غالبا بـ 1).

إنقاص decrement: تحديث يُنقص من قيمة المتغير.

تكرار iteration: تنفيذ متكرر لمجموعة من العبارات باستخدام الاجترار أو الحلقات.

حلقة مفرغة infinite loop: حلقة لا توفّى بها شروط الاقفال.

7.9 تمارين

تمرين 7.3 لتفحص خوارزمية الجذر التربيعي الانف ذكرها في هذا الفصل، بإمكانك مقارنتها بالاقتران الجاهز `math.sqrt` اكتب اقترانا اسمه `test_square_root` يطبع جدولا كهذا:

```
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
```

```
8.0 2.82842712475 2.82842712475      4.4408920985e-16
9.0 3.0           3.0                0.0
```

العمود الاول عدد ، a، الثاني يكون الجذر التربيعي محسوبا باستخدام الاقتران من القسم 7.5، و الثالث يكون الجذر التربيعي محسوبا باستخدام `math.sqrt` ، و العمود الرابع هو القيمة المطلقة للفرق بين التوقعين.

تمرين 7.4 الاقتران الجاهز `eval` يأخذ محارف و يقيمها باستخدام مفسّر بايثون، مثلا:

```
>>> eval('1 + 2 * 3')
7
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<type 'float'>
```

اكتب اقترانا اسمه `eval_loop` يكرر طلب الادخال من المستخدم ثم يأخذ المدخلات و يقيمها باستخدام `eval` و يطبع نتيجة التقييم.

على الاقتران الاستمرار حتى يطبع المستخدم done، ثم يرجعة قيمة اخر تعبير قيمه

تمرين 75 الرياضياتي سرينيفازا رامانوجان وجد متسلسلة لانهائية يمكن استعمالها لإيجاد تقريب لـ 1/ط:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{(4k)!\,(1103+26390K)}{(k!)^4 396^{4k}}$$

اكتب اقترانا اسمه `estimate_pi` يستخدم هذه المعادلة لحساب و ارجاع تقريب لـ ط يجب استخدام `while` لحساب المجاميع حتى يصبح اخر مجموع أقل من `1e-15` (بايثون يكتبها هكذا للتعبير عن 10^{-15}) يمكنك مقارنة التنيجة بـ `math.pi`.

الحل: `http://thinkpython.com/code/pi.py`.

الفصل الثامن

المحارف

8.1 المحارف هي تسلسلات

المحارف (الطلاسم إن أردت) هي سلاسل من الاشكال و الحروف و الارقام، و يمكنك الوصول إلى كل من الحروف (أو الاشكال و الارقام) بمؤثر الحاصرة[]:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

تختار العبارة الثانية الشكل الاول من `fruit` ثم تعينه لـ `letter`.

التعبير الذي بين الحاصرتين يدعى المؤشر index، يؤشر المؤشر إلى أي من حروف السلسلة تريد (و منه الاسم).

إلا أنك قد لا تحصل ما توقعت:

```
>>> letter = fruit[0]
>>> print letter
b
```

إذن فـ `b` هي الحرف الصفر من `banana` , و كذلك فـ `n` هو الحرف الثاني.

بإمكانك استخدام أي تعبير كمؤشر، بما فيها المتغيرات و العوامل لكن يجب أن تكون قيمة المؤشر عدد صحيح و إلا فستحصل على خطأ:

```
>>> letter = fruit[15]
TypeError: string indices must be integers, not float
```

8.2 len

الاقتران الجاهز `len` يرجع عدد حروف المحارف:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

فلكي تحصل على الرقم الاخير في المحارف قد تغريك الكتابة كالتالي:

```
length = len(fruit)
last = fruit[length]
IndexError: string index out of range
```

السبب في هذا الخطأ `IndexError` هو أنه لا يوجد حرف في `banana` رقم مؤشره 6، فبما أننا ابتدأنا العد بصفر فسننتهي بـ 5، إذن لكي تحصل على الحرف الاخير عليك طرح 1 من `length`:

```
>>> last = fruit[length-1]
```

```
>>> print last
a
```

أو يمكنك، كبديل، استخدام أرقام المؤشرات السالبة، و التي تبدأ العد عكسيا من نهاية المحارف إلى بدايتها فالتعبير `fruit[-1]` هو اخر حرف و `fruit[-2]`هو الحرف قبل الاخير، و هكذا.

8.3 المرور باستخدام for

كثيرة هي العمليات الحوسبية التي بها تتم معالجة المحارف حرفا تلو الآخر، هذه العمليات عادةً ما تبدا في بداية المحارف ثم تختار كل حرف بدوره، تفعل شيئا ما بهذا الحرف، ثم تستمر حتى النهاية. هذا النمط من المعالجات يسمى المرور **traversal**، حلقة `while` هي احدى طرق كتابة المرور:

```
index = 0
while index < len(fruit):
   letter = fruit[index]
   print letter
   index = index + 1
```

تمر هذه الحلقة في المحارف ثم تعرض كل حرف على سطر لوحده، شرط هذه الحلقة كان `index < len(fruit)`، فعندما تتساوى قيمة `index` مع طول المحارف تكون نتيجة الشرط `false` و لن ينفذ متن الحلقة، اخر حرف تصله الحلقة هو الحرف الذي مؤشره `len(fruit)-1`، و هو اخر حرف في السلسلة.

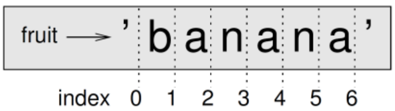
تمرين 8.1 اكتب اقترانا يأخذ محارف كقرينة ثم يعرض حروفها معكوسة و كل حرف في سطر.

طريقة أخرى لكتابة حلقة مرور هي عن طريق حلقة `for`:

```
for char in fruit:
   print char
```

في كل دورة في الحلقة فإن الحرف التالي سيعين للمتغير `char` و سيستمر التدوير إلى أن تنتهي كل الحروف.

المثال التالي يبين طريقة اضافة محارف و حلقة `for` لتوليد أَلِفبائية (حسب التسلسل الابجدي). في كتاب "افسح الطريق للبطيطات" لروبرت مكلوسكي، كانت أسماء البط Jack, Lack, Mack, Nack , Pack و Quack، هذه الحلقة تُخرج اسماءها بالترتيب:



الشكل 8.1: مؤشرات شرائح

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
   print letter + suffix
```

المُخرَج هو:

```
Jack
Kack
Lack
```

```
Mack
Nack
Oack
Pack
Qack
```

طبعا هذا غير صحيح لأن تهجئة Ouack و Quack غير صحيحة.

تمرين 8.2 عدل البرنامج لتصحح الخطأ.

8.4 شرائح المحارف

أي قطعة من محارف تسمى شريحة slice، اختيار شريحة يشبه اختيار حرف:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:12]
Python
```

العامل `[n:m]` يرجع جزء من المحارف مبتدءا بالحرف الذي في الموقع n و حتى الحرف الذي في الموقع m متضمنا الحرف الاول و لكن ليس الاخير، هذا التصرف لا يتلاءم مع البديهة، لتسهيل الامر تخيل أن المؤشرات تشير إلى ما بين مواقع الحروف، كما في الشكل 8.1.

ان لم تكتب قيمة المؤشر الاول (الذي قبل النقطتين)، يبدأ اختيار الشريحة من بداية المحارف، و إن لم تكتب المؤشر الاخير تكون نهاية الشريحة هي نهاية المحارف:

```
>>> fruit = 'banana'
>>> fruit[:3]
ban
>>> fruit[3:]
ana
```

إن كانت قيمة المؤشر الاول أكبر من أو تساوي الثاني فالنتيجة هي محارف فارغة، و ستتمثل بعلامتي اقتباس:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

المحارف الفارغة ليس بها حروف و طولها 0 ، لكن عدا ذلك فهي تماما كأي محارف أخرى.

تمرين 8.3 بما أن `fruit` هي محارف، فما معنى `fruit[:]` ؟

8.5 المحارف ثبيتةٌ لا تُعدَّل

من المغري استخدام `[]` إلى يسار التعيين بنيّة تغيير حرف في المحارف:

```
>>> greeting = 'Hello, World!'
greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

الكائن هنا هو المحارف، و **العنصر** هو الحرف الذي حاولت تعيينه. في الوقت الحاضر سنعتبر الكائن هو نفسه القيمة، لكننا سنهذب هذا التعريف لاحقا، العنصر هو أحد القيم في تسلسل ما.

سبب الخطأ هو أن المحارف لا تعدَّل، أي أنه ليس بالامكان تغيير محارف موجودة. البديل لهذا هو إيجاد محارف جديدة ثم

التعديل عليها:

```
>>> greeting = 'Hello, World!'
>>> new_greeting = 'J' + greeting[1:]
print new_greeting
Jello, World!
```

أضاف هذا المثال حرف أول جديد لشريحة من `greeting` و لم تؤثر في المحارف الاصلية.

8.6 البحث

مالذي يفعله الاقتران التالي:

```
def find(word, letter):
   index = 0
   while index < len(word):
      if word[index] == letter:
         return index
      index = index + 1
   return -1
```

من حيث المعنى فـ `find` هي المقابل للرمز `[]` ، فبدلا من أخذ مؤشر ثم انتزاع الحرف المقابل له، تأخذ `find` حرفا و تجد المؤشر الذي عنده يظهر الحرف. إن لم يوجد الحرف فسيُرجع الاقتران 1-.

كان هذا المثال الاول حيث نرى عبارة `return` داخل حلقة، فإن تصادف أن `word[index] == letter` فإن الاقتران سيخرج من الحلقة و يرجع فورا.

إن لم يظهر الحرف في المحارف فإن البرنامج سينتهي بشكل عادي و يرجع القيمة 1-.

هذا النمط من الحوسبة – مرور على تسلسل و الرجوع عندما نجد ما نبحث عنه – يسمى البحث search

تمرين 8.4 عدل `find` بحيث يصبح لها برمتر ثالث يكون مؤشرافي `word` يحدد أين سيبدأ البحث.

8.7 التدوير و العد

يَعُدُّ البرنامج التالي عدد المرات التي يظهر فيها الحرف a في محارف:

```
word = 'banana'
count = 0
for letter in word:
   if letter == 'a':
      count = count + 1
print count
```

يوضح هذا المثال نمط اخر من انماط الحوسبة يسمى العداد **counter**. يُستهل المتغير `count` بصفر ثم تزداد قيمته في كل مرة يجد فيها `a` و عندما تُترك الحلقة، يحتفظ `count` بالنتيجة – و التي هي عدد `a` في السلسلة.

تمرين 8.5 كبسل النص البرمجي في اقتران و سمِّه `count` ، ثم عممه بحيث يأخذ المحارف و الحرف كقرينتين.

تمرين 8.6 أعد كتابة هذا الاقتران بحيث أنه بدلا من المرور في المحارف فإنه يستخدم النسخة ذات الثلاثة برمترات في `find` في القسم السابق.

8.8 طرق المحارف

الطريقة (method) تشبه الاقتران – تأخذ قرائن و ترجع قيمة – الا أن نحوها يختلف. فمثلا، الطريقة upper تأخذ المحارف و ترجع محارف جديدة جاعلة كل حروفها كبيرة:

و بدلا من من استخدام بناء الاقتران upper(word)، فإن كتابة الطرق تكون هكذا word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

هذا النمط من التنويت بالنقاط **dot notation** يحدد اسم الطريقة ، upper ، و اسم المحارف التي ستطبق الطريقة عليها، word. الاقواس الفارغة تعني أن هذه الطريقة لا تأخذ قرائن.

نداء الطريقة يدعى استدعاء **invocation** (نداء الاقتران يسمى call) ، و في هذه الحالة نقول: نحن نستدعي upper على word.

تبين أن هناك طريقة للمحارف تسمى find، و تقوم بنفس عمل الاقتران الذي كتبناه للتو:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

في هذا المثال استدعينا find على word و مررنا الحرف الذي نبحث عنه كبرمتر.

الحق أن find أعم من الاقتران الذي كتبناه، فبوسعها ايجاد أجزاء من المحارف:

```
>>> word.find('na')
2
```

و بوسعها أيضا أخذ المؤشر، حيث يبدأ البحث كقرينة ثانية:

```
>>> word.find('na', 3)
4
```

و كقرينة ثالثة تأخذ المؤشر حيث ينتهي البحث:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

فشل هذا البحث لأن b لا تظهر في النطاق من 1 إلى 2 (لا يتضمن 2).

تمرين 8.7 هناك طريقة للمحارف تسمى count و هي شبيهة بالاقتران الذي كتبناه في التمرين السابق. اقرأ وثائقها ثم اكتب استدعاءا يرجع عدد a في banana.

تمرين 8.8 اقرأ وثائق طرق المحارف فقد يفيدك التدرب على بعضها لتتعلم كيف تستخدم، هناك فائدة مخصوصة لـ strip و replace.

الوثائق تستخدم نحوا قد يكون مشوشا، مثلا في find(sub[, start[, end]])، الاقواس تعني قرائن اختيارية، فـ sub مطلوبة لكن start اختيارية، و إن ضُمّنت start تصبح end اختيارية.

8.9 المؤثر in

الكلمة in هي رمز بوليان، تأخذ محارفين و ترجع إما True إن ظهرت المحارف الاولى كجزء من الثانية:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

مثال: الاقتران التالي يطبع كل حروف word1 التي تظهر في word2:

```
def in_both(word1, word2):
   for letter in word1:
      if letter in word2:
         print letter
```

إن أُحسن اختيار أسماء المتغيرات فإن بايثون يُقرأ كما تقرأ الانجليزية، يمكن قراءة هذه الحلقة كالتالي:

"for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter"

هذا ما ستحصل عليه ان قارنت التفاح بالبرتقال:

```
in_both('apples', 'oranges')
a
e
s
```

8.10 مقارنة المحارف

مؤثرات النسبة تعمل على المحارف. لكي نرى ان كانت محارفتين متساويتين:

```
if word == 'banana':
   print 'All right, bananas'
```

مؤثرات اخرى مفيدة لوضع الحروف في ترتيب ابجدي:

```
if word < 'banana':
   print 'Your word,' + word + ', comes before banana'
elif word > 'banana':
   print 'Your word,' + word + ', comes after banana'
else:
   print 'All right, bananas'
```

لا يتداول بايثون الاحرف الكبيرة و الاحرف الصغيرة كما يتداولها الناس. كل الحروف الكبيرة تأتي قبل الصغيرة، فإن:

Your word, Pineapple تأتي قبل before, banana

من طرق التعامل مع هذه المشكلة، تحويل كل الحروف إما إلى صغيرة أو كبيرة قبل المقارنة. انقش هذه المعلومة في رأسك، ستفيدك إن واجهت رجلا مسلحا بأناناسة.

8.11 علاج الاخطاء

عندما تستخدم المؤشرات indices للمرور في قيم في تسلسل، يكون التعرف على بداية و نهاية المرور مخادعا. التالي اقتران يفترض أنه يقارن بين كلمتين و يرجع `True` ان كانت احداها معكوس الاخرى، الا أن به خطأين:

```
def is_reverse(word1, word2):
   if len(word1) != len(word2):
      return False

   i = 0
   j = len(word2)

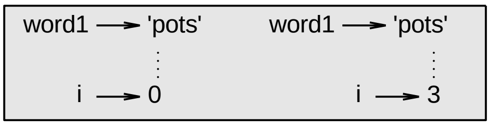
   while j > 0:
      if word1[i] != word2[j]:
         return False
      i = i+1
      j = j-1
   return True
```

عبارة `if` الاولى تفحص اذا ما كان للكلمتين نفس الطول، ان كان لا، سترجع `False` مباشرة ، بعدها، و إلى نهاية الاقتران، يمكننا افتراض ان الكلمات متساوية في الطول هذا مثال على نمط الوصي في كتابة النصوص البرمجية كما في القسم 6.8.

`i` و `j` مؤشرين: تجتاز `i` السلسلة `word1` قُدُما بينما تجتاز `j` السلسلة `word2` رجوعا، إن وجدنا حرفين غير متشابهين سنرجع `False` مباشرة، أما ان اجتزنا الحلقة كاملة و كل الحروف متشابهة فسنرجع `True`.

ان امتحنا هذا الاقتران بكلمتي "pots" و "stop" سنتوقع الحصول على `True`، الا أننا سنحصل على IndexError:

```
>>> is_reverse('pots', 'stop')
...
```



الشكل 8.2: رسم الحالة

```
File "reverse.py", line 15, in is_reverse
if word1[i] != word2[j]:
IndexError: string index out of range
```

لعلاج خطأ كهذا، خطوتي الاولى ستكون طباعة قيم المؤشرات مباشرة قبل موقع ظهور الخطأ:

```
   while j > 0:
      print i, j        # print here

      if word1[i] != word2[j]:
         return False
      i = i+1
      j = j-1
```

عندما اشغل البرنامج الان سأحصل على معلومات أكثر:

```
>>> is_reverse('pots', 'stop')
```

```
0 4
...
IndexError: string index out of range
```

خلال أول دورة في الحلقة كانت قيمة j هي 4 ، و هي بالفعل خارج نطاق المحارف pots، فقيمة المؤشر لآخر حرف هي 3 ، و عليه فالقيمة الابتدائية للمؤشر j يجب أن تكون len(word2)-1.

ان أصلحت هذا الخطأ و شغلت البرنامج سأحصل على:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

في هذه المرة حصلنا على الجواب الصحيح، لكن يبدو أننا مررنا في الحلقة ثلاث مرات فقط، و هذا مثير للشك. لكي نستوضح ما الذي يحدث، يفيدنا رسم حالة للبرنامج، خلال التكرار الاول يظهر إطار is_reverse في الشكل 8.2.

أخذت بعض الحرية في ترتيب المتغيرات في الاطار و وضع اسهم منقطة لبيان أن قيم i و j تشير لحروف في word1 و word2.

تمرين 8.9 انطلاقا من هذا الرسم، نفذ البرنامج على الورق مغيرا قيم i و j في كل تكرار جد و صحح الخطأ في هذا الاقتران.

8.12 المعاني

كائن object: أي شيء قد يشير اليه متغير بإمكانك في الوقت الحالي استعمال قيمة و كائن للتعبير عن نفس الشيء.

تسلسل sequence: مجموعة مرتبة، أي: مجموعة من القيم مرتبة بحيث يكون لكل قيمة عدد صحيح كمؤشر.

عنصر item: إحدى القيم في مجموعة مرتبة.

مؤشر index: عدد صحيح يستعمل لاختيار عنصر في مجموعة مرتبة.

شريحة slice : جزء من محارف يحددها نطاق المؤشرات.

سلسلة فارغة empty string: محارف ليس بها حروف و طولها صفر، تمثلها علامتي اقتباس.

غير متبدل immutable: صفة التسلسل الذي لا يمكن التعيين لعناصره.

مرور traverse: القيام بنفس العملية لكل العناصر في تسلسل.

بحث search: نمط من أنماط المرور، بحيث تتوقف العملية عندما تجد ما تبحث عنه.

عداد counter : متغير يستخدم لعد شيء ما عادة يستهل بقيمة 0.

طريقة method : اقتران مرتبط بكائن يستدعى بنمط التدوين بالنقاط.

استدعاء invocation: عبارة لنداء الطريقة.

8.14 تمارين

تمرين 8.10 يوسع شريحة المحارف أن تأخذ مؤشر ثالث اسمه "*step size*" و هو عدد الامكنة بين الحروف المتتالية، فإن كانت تساوي 2 سيعني حرف نعم و حرف لا، و 3 تعني حرف نعم و حرفين لا...،لخ

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

و *step size* قيمتها 1- ستخترق كل الكلمة عكسيا، و هكذا فـ `[::-1]` ستولد محارف معكوسة.

استخدم هذا المعنى لكتابة نص برمجي من سطر واحد بدلا من `is_palindrome` في التمرين 6.6

تمرين 8.11 جميع الاقترانات التالية تهدف إلى التحقق إذا ما كانت المحارف تحتوي على حروف صغيرة، إلا أن بعضها خطأ اشرح كل اقتران منها (على فرض أن البرمتر محارف).

```
def any_lowercase1(s):
   for c in s:
      if c.islower():
         return True
      else:
         return False

def any_lowercase2(s):
   for c in s:
      if 'c'.islower():
         return 'True'
      else:
         return 'False'

def any_lowercase3(s):
   for c in s:
      flag = c.islower()
   return flag

def any_lowercase4(s):
   flag = False
   for c in s:
      flag = flag or c.islower()
   return flag

def any_lowercase5(s):
   for c in s:
      if not c.islower():
         return False
   return True
```

تمرين 8.12 نظام التشفير *ROT13* كان نظاما ضعيفا، إعتمد على "تدوير" الحروف في كل كلمة 13 منزلة، التدوير يعني إزاحة الحرف في الأبجدية، و حتى المرور مرة أخرى- إن تطلب الأمر- في بداية الأبجدية. فإن بدلنا A بما يليها بثلاثة حروف ستصبح D مثلا و إن بدلنا Z بما يليها بواحد تصبح A.

اكتب اقترانا و سمه `rotate_word` يأخذ محارف و عدد صحيح كبرمترات، ثم يرجع محارف فها المحارف الاصلية بعد أن دورت حسب العدد المعطى.

مثلا، "cheer" دوّرت بـ 7 ستصبح "jolly" و "melon" إن دورت بـ 10- ستصبح "cubed".

قد يخدمك استخدام الاقتران الجاهز `ord` فهو يحول الحرف إلى رمزه الرقمي و `chr` التي تحول الرمز الرقمي إلى حرف.
بعض النكات المسيئة على الانترنت مكتوبة بـ *ROT13* فإن كنت تحتمل المزاح الثقيل جِدها و فك تشفيرها.
الحل: `http://thinkpython.com/code/rotate.py`.

الفصل التاسع

دراسة حالة: لعبة كلمات

9.1 قراءة قوائم الكلمات

لتطبيق التمارين في هذا الفصل سنحتاج إلى قائمة بالكلمات الانجليزية، هذه القوائم متوفرة بكثرة على الانترنت. لكن أكثرها ملاءمة لهدفنا هي احدى القوائم التي جمعها و ساهم بها للملك العام Grady Ward، و كانت جزءا من مشروع "موبي لكسكون" أنظر http://wikipedia.org/wiki/Moby_Project.

هي قائمة من 113809 من كلمات العاب الكلمات المتقاطعة الرسمية، أي ان الكلمات التي تحتويها تعتبر مقبولة في عمل الكلمات المتقاطعة و غيرها من العاب الكلمات، اسم الملف في مجموعة موبي هو 113809of.fic بإمكانك تحميل نسخة باسم ابسط word.txt من http://thinkpython.com/code/words.txt.

هذا الملف نص خام، مما يعني أنه يمكنك فتحه بأي معالج نصوص، لكن بإمكانك قراءته من بايثون أيضا، فالاقتران الجاهز open يأخذ اسم الملف كبرمتر و يرجع كائن ملف:

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

fin هي الاسم الشائع لكائن الملف، و عندما تَستخدم الوضع r سيعني أن الملف فُتِح للقراءة (على عكس w فُتِح للكتابة).

لقراءة كائن الملف طرق عديدة ، منها readline التي تقرأ الحروف من الملف حتى تصل إلى السطر التالي، ثم تُرجع النتيجة على شكل محارف:

```
>>> fin.readline()
'aa\rr\n'
```

الكلمة الاولى في هذه القائمة هذه هي "aa" و التسلسل \r\n يمثل حرفان فارغان (مسافات بيضاء) : الرجوع و السطر الجديد، الذان يفصلان هذه الكلمة عن تاليتها.

و كائن الملف هذا لا ينسى المكان الذي وصل إليه في الملف، فإن استدعيت readline مرة ثانية سيعرف من أين يكمل و سيرجع لك الكلمة التالية:

```
>>> fin.readline()
'aah\r\n'
```

الكلمة التالية هي aah (و هي كلمة جائزة تماما، فتوقف عن النظر إليّ هكذا!)، إلا إن كان ما يزعجك هو المسافة البيضاء

فالتخلص منها مقدور عليه بالطريقة strip:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> print word
aahed
```

يمكنك استخدام كائن الملف كجزء من حلقة for ، البرنامج التالي يقرأ words.txt و يطبع كل كلمة فيه على سطر:

```
fin = open('words.txt')
for line in fin:
   word = line.strip()
   print word
```

تمرين 9.1 اكتب برنامجا يقرأ words.txt ثم يطبع الكلمات التي يزيد عدد حروفها على العشرين (دون احتساب المسافات البيضاء).

9.2 تمارين

حلول هذه التمارين متوفرة في القسم التالي، لكن عليك على الاقل محاولة حلها قبل النظر إلى الحلول.

تمرين9.2 في العام 1939 كتب إرنست فنسنت رايت رواية من 50000 كلمة و سماها Gadsby، و كانت لا تحتوي على الحرف e، و بما أن الحرف e هو أكثر حروف الانجليزية استعمالا، لذلك فإن ما قام به ليس بالامر الهين.

"In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow going at first, but with caution and hours of training you can gradually gain facility.

حسنا سأكتفي الان

اكتب اقترانا و سمِه has_no_e يرجع True إن لم تحتو الكلمة المدخلة على الحرف e.

عدل البرنامج من القسم السابق بحيث يطبع الكلمات التي ليس بها e فقط، ثم يحسب النسبة المئوية للكلمات التي في قائمة ليس بها e.

تمرين 9.3 أكتب اقترانا و سمه avoids يأخذ كلمة و محارف من الحروف الممنوعة، ثم يرجع True ان كانت الكلمة لا تحتوي على أي من الحروف الممنوعة.

عدل برنامجك بحيث يطلب من المستخدم ادخال المحارف الممنوعة ثم يطبع الكلمات التي لا تحتوي على أي منها، هل تستطيع ايجاد تركيبة من 5 حروف ممنوعة بحيث تستثني أقل عدد من الكلمات؟

تمرين 9.4 اكتب اقترانا و سمه uses_only يأخذ الكلمة كمحارف ثم يرجع True ان كانت الكلمة تستعمل فقط الحروف التي في القائمة، هل يمكنك بناء جملة تستخدم الحروف acefhlo؟ جملة غير Hoe alfalfa.

تمرين 9.5 اكتب اقترانا و سمه uses_all يأخذ كلمة و محارف مطلوبة، ثم يرجع True ان كانت الكلمة تستخدم جميع الحروف المطلوبة على الاقل مرة واحدة. كم عدد الكلمات التي تستخدم جميع حروف العلة aeiou؟ و ماذا عن aeiouy؟

تمرين 9.6 اكتب اقترانا اسمه is_abecedarian يرجع True إن كانت حروف الكلمة تظهر بالترتيب الابجدي (لا

مشكلة بالاحرف المكررة). كم عدد الكلمات التي ينطبق عليها هذا الحال؟

9.3 البحث Search:

هنالك أمر مشترك بين جميع التمارين السابقة، فيمكن حلها جميعا بنمط البحث الذي رأيناه في القسم 8.6 ابسط مثال هو:

```
def has_no_e(word):
   for letter in word:
      if letter == 'e':
      return False
   return True
```

حلقة `for` تمر على الحروف في `word`. و إن وجدت e ستُرجع `False` مباشرة، و الا فسننتقل إلى الحرف التالي و ان خرجنا من الحلقة بشكل عادي فسيعني أننا لم نجد e و عليه يكون المُرجع `True`.

`avoid` نسخة معممة من أكثر من `has_no_e` لكن لها نفس البنيان:

```
def avoids(word, forbidden):
   for letter in word:
      if letter in forbidden:
         return False
   return True
```

بدلا من قائمة الحروف الممنوعة لدينا قائمة من حروف مسموحة، فإن وجدنا حرفا في `word` ليس `available` سنرجع `False`.

`Usese_all` مشابهة الا أننا نعكس دور الكلمات و المحارف:

```
def uses_all(word, required):
   for letter in required:
      if letter not in word:
         return False
   return True
```

بدلا من المرور على حروف `word` تمر الحلقة على الحروف المطلوبة، ان لم تظهر في الكلمة أي من الحروف المطلوبة فسنرجع `False`.

لو كنت تفكر كعالم حاسوب حقا، لتنبهت إلى ان `uses_all` هي مظهر آخر لمسألة حُلت من قبل، و لكنت قد كتبت:

```
def uses_all(word, required):
   return uses_only(required, word)
```

هذا مثال على اسلوب لتطوير البرامج يسمى تَذكُّر المشكلة **problem recognition**، و يعني أن المشكلة التي بين يديك ذكرتك بمشكلة حللتها من قبل، ثم تطبق الحل الذي طور سابقا على المشكلة الحالية

9.3 التدوير بالمؤشرات

كَتبتُ الاقترانات في القسم السابق بحلقات `for` لأنني احتجت إلى حروف المحارف و حسب، المؤشرات لم تعنيني.

لكن في `is_abecedarian` علينا مقارنة حرفيين متجاورين، و هذا، باستخدام حلقة `for`، أمر مخادع:

```
def is_abecedarian(word):
   previous = word[0]
   for c in word:
      if c < previous:
         return False
      previous = c
```

```
   return True
```

البديل هو استخدام الاجترار:

```
def is_abecedarian(word):
   if len(word) <= 1:
      return True
   if word[0] > word[1]:
      return False
   return is_abecedarian(word[1:])
```

و بديل آخر هو استخدام حلقة while:

```
def is_abecedarian(word):
   i = 0
   while i < len(word)-1:
      if word[i+1] < word[i]:
         return False
      i = i+1
   return True
```

تبدأ الحلقة بـ i=0 و تنتهي بـ i=len(word)-1، في كل مرة تدور في الحلقة ستقارن الحرف في المكان *i* (يمكنك اعتبار المكان *i* المكان الحالي) بالحرف في المكان i+1 (يمكنك اعتباره المكان التالي).

فإن كان الحرف التالي أصغر (اسبق ابجديا) من الحرف الحالي فنكون قد اكتشفنا ثغرة في الالفبائية و نرجع عندها False.

و ان انتهينا من التدوير دون ان نجد خطأ فمعناه ان الكلمة قد اجتازت الاختبار. و لتقنع نفسك بأن الحلقة انتهت بشكل صحيح، خذ مثالا كـ flossy طول الكلمة 6 فتكون أخر دورة في الحلقة عندما تكون *i* = 4، و هي المؤشر للحرف الثاني قبل الاخير. في التكرار الاخير تقارن الحرف الثاني قبل الاخير بالحرف قبل الاخير، و هو مانريده.

هذه نسخة من is_palidrome (انظر التمرين 6.6) تستخدم مؤشرين، يبدأ الاول في المقدمة و ينطلق صعودا و يبدأ الثاني في النهاية و ينطلق هبوطا.

```
def is_palindrome(word):
   i = 0
   j = len(word)-1
   while i<j:
      if word[i] != word[j]:
         return False
      i = i+1
      j = j-1
   return True
```

أو، ان ذكرك هذا بمشكلة حلت من قبل، لكنت كتبت:

```
def is_palindrome(word):
   return is_reverse(word, word)
```

هذا على اعتبار أنك حللت التمرين 8.9

9.5 علاج الاخطاء

اختبار البرامج صعب، كانت الاقترانات في هذا القسم سهلة نسبيا لأنك تستطيع فحص النتائج يدويا. و ان يكن، فاختيار

كلمات تستطيع اختبار كل احتمالات الاخطاء شيء يقع بين الصعب و المستحيل.

بأخذ `has_no_e` كمثال، هناك حالتان واضحتان يكن فحصها: فالكلمات التي بها e ترجع `False` و الاخرى ترجع `True`، لن يتعبك ايجاد كلمة لكليهما.

في كل من الحالتين هناك حالات فرعية اخرى اقل وضوحا، ففي الكلمات التي تحتوي على e عليك اختبار أيٍ منها تكون فيها e في الاول ثم في الوسط ثم في نهاية الكلمة. ثم عليك اختبار الكلمات الطويلة و القصيرة و القصيرة جدا (كالمحارف الفارغة) المحارف الفارغة مثال على الحالات الخاصة، و التي هي احدى الحالات غير الواضحة التي يعشش فيها الخطأ.

اضافة لحالات الاختبار التي اوجدتها، يمكنك فحص برنامجك بقوائم الكلمات مثل `words.txt`. بالنظر في المخرجات يمكنك التقاط الاخطاء، لكن عليك الحذر: فقد تتمكن من التقاط نوع من الأخطاء (كالكلمات التي يجب الا تُضمَّن، الا أنها مضمنة) و لا تلتقط النوع الاخر (كالكلمات التي يجب ان تكون مضمنة و لكنها ليست كذلك).

في العموم، الاختبار قد يساعدك في التقاط الاخطاء، الا أنه من الصعب ايجاد مجموعة متكاملة من حالات الاختبار، و إن وجدت، فليس بوسعك التأكد من أن البرنامج صحيح.

> اختبار البرامج يمكّننا من إظهار البق إن كان موجودا، لكنه لا يمكننا من إظهار عدم وجوده!
>
> - إدسغر و. داكسترا

9.6 المعاني

كائن ملف file object: قيمة تمثل ملفا مفتوحا.

تذكر المشكلة problem recognition: طريقة لحل المشاكل بالتعبير عنها كشبيهة لمشكلة حلت سابقا.

حالة خاصة special case: حالة اختبار غير نمطية أو غير واضحة (و احتمال التعامل معها بشكل صحيح صغير).

9.7 تمارين

تمرين 97 السؤال مبني على احجية أذيعت على برنامج "حديث السيارة" Car Talk:

`http://www.cartalk.com/content/puzzlers`.

> "أعطني كلمة بها ثلاثة حروف مشددة متتالية double letters. اليك بعض الكلمات التي ينطبق عليها الشرط تقريبا. مثلا *c-o-m-m-i-t-t-e-e* قد تناسب لولا الـ *i* التي تشوهها أو *m-i-s-s-i-s-s-p-p-i* التي ستكون رائعة لولا الـ *i* . الا أنه هنالك كلمة بها ثلاثة حروف مشددة متتالية بالفعل، و على حد علمي هي الكلمة الوحيدة. بالطبع يمكن أن تكون هنالك 500 كلمة أخرى الا أنني قلت على حد علمي. ما هي الكلمة؟

اكتب برنامجا يجدها. الحل: `http://thinkpython.com/code/cartalk1.py`.

تمرين 9.8 هذه احجية اخرى من Car Talk `http://wwwcartalk.com/content/puzzlers`.

> "كنت أسوق ذات يوم، و حدث أن لفت انتباهي عداد المسافة في سيارتي، للعداد 6 خانات و

كلها للأميال الكاملة فقط فإن كانت سيارتي عند 300000 ميل مثلا، سأرى 0-0-0-0-0-3 .

الآن، ما رأيته ذلك اليوم كان مثيرا. لاحظت بأن الخانات الاربعة الاخيرة كانت متناظرة، أي تقرأ من اليمين إلى اليسار كما تقرأ من اليسار إلى اليمين، مثل 5-4-4-5 هي متناظرة، فإن كانت هي التي على العداد ستكون قراءته 5-4-4-5-1-3.

بعد ميل واحد، أصبحت الخانات الخمسة الاخيرة متناظرة، مثل 6-5-4-5-6-3، ثم بعد ميل اخر اصبحت الخانات الاربعة الوسطى متناظرة. استعد الان: بعد ميل اخر اصبحت الخانات الستة جميعها متناظرة!"

"السؤال هو: ماذا كانت قراءة العداد عندما لاحظته في المرة الاولى؟"

اكتب برنامجا يفحص كل الارقام ذات الستة خانات و يطبع أي رقم يفي هذه المتطلبات.

الحل: `http://thinkpython.com/code/cartalk2.py`.

تمرين 9.9 أيضا من Car Talk بإمكانك حله بالبحث.

`http://www.cartalk.com/content/puzzlers`.

" كانت لدي مؤخرا زيارة مع الوالدة، و انتبهنا إلى أن الخانتين التين تكونان عمري هما عمر الولدة ان عُكسا. مثلا إن كان عمرها 73 سأكون 37. ثم تساءلنا عن عدد المرات التي حدث فيها هذا طوال عمرينا، الا أن الحديث انتقل لموضوع آخر فلم نصل للنتيجة.

عند عودتي للبيت، اكتشفت أن خانات عمرينا كانت المعكوس ستة مرات حتى الان، ثم اكتشفت أنه ان طال عمرينا فسيحدث نفس الامر بعد بضعة سنوات، و إن طال أكثر فسيحدث مرة أخرى، بكلمات اخرى فإنه قد يحدث ثماني مرات فالسؤال هو: ما هو عمري الان؟"

اكتب برنامج بايثون يبحث عن حل لهذه المسألة تلميح: قد تكون طريقة zfill لسلاسل الحروف مفيدة

الحل: `http://thinkpython.com/code/cartalk3.py`.

الفصل العاشر

القوائم

10.1 القائمة هي تسلسل

مثل المحارف، فالقائمة هي تسلسل لقيم. كانت القيم في المحارف حروفا (حروف و أرقام و أشكال)، أما في القائمة فيمكن للقيم أن تكون أي شيء. القيم في القائمة تدعى عناصر elements و أحيانا items

هنالك عدة طرق لإيجاد قائمة جديدة، الاسهل حصر العناصر بين قوسين مربعين ([and]) :

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

المثال الاول قائمة بها أربعة أعداد صحيحة، و الثاني قائمة بثلاثة محارف. ليس شرطا أن تكون عناصر القائمة من نفس النوع. القائمة التالية تحتوي على محارف و قيمة حقيقية و عدد صحيح و (ها!) قائمة أخرى:

```
['spam', 2.0, 5, [10, 20]]
```

القائمة التي في داخل اخرى تسمى (**معششة**) nested.

القائمة التي ليس بها عناصر تدعى قائمة فارغة، و يمكنك عملها بقوسين مربعين فارغين [].

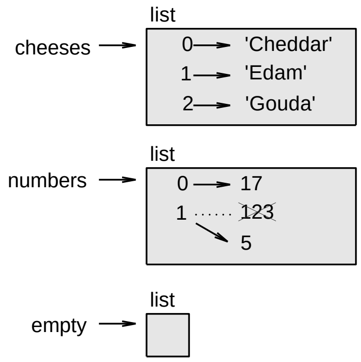
و كما تتوقع فبإمكانك تعيين قيم القائمة لمتغيرات:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2 القوائم ليست ثبيتةً، تعدَّل

نحوُ عبارة الوصول لأحد عناصر القائمة هو نفسه نحو الوصول إلى حروف المحارف- مؤثر القوسين المربعين []. التعبير بينهما يحدد المؤشر. تذكّر بأن المؤشرات تبدأ بـ 0:

```
>>> print cheeses[0]
Cheddar
```



الشكل 10.1: رسم الحالة

و على خلاف المحارف فإن القوائم تُعدَّل، فعندما يظهر مؤثر القوسان على يسار التعيين فإنه يحدد العنصر الذي سنعين له:

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

العنصر ذو المؤشر 1 في قائمة `numbers` و الذي كان 123 أصبح الان 5.

بإمكانك اعتبار القوائم كعلاقة بين مؤشرات و عناصر، هذه العلاقة تسمى **mapping** أي خطُّ الخطوط أو الخرائط ، فكل مؤشر "يرسم خطا" يرشد إلى أحد العناصر. الشكل 10.1 يظهر رسم الحالة لـ `cheeses, number,` و `empty`:

القوائم ممثلة بصناديق باسم "list" في الخارج و العناصر في الداخل `cheeses` تشير إلى قائمة من ثلاثة عناصر مؤشرة بـ 1، 2 و 3. تحتوي `numbers` على عنصرين، و يبين الرسم أن القيمة المعينة للعنصر الثاني أعيد تعيينها من 123 إلى 5 `empty` تشير إلى قائمة بلا عناصر.

مؤشرات القوائم تعمل كمؤشرات المحارف:

- أي عدد صحيح أو تعبير ينتجه يمكن استخدامه كمؤشر.
- إن حاولت قراءة أو كتابة عنصر غير موجود ستحصل على خطأ `IndexError`.
- إن كانت قيمة المؤشر سالبة سيكون العد عكسيا (من نهاية القائمة).

كذلك فالمؤثر `in` يقوم بنفس العمل.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 المرور على عناصر القائمة

الطريقة الشائعة للمرور هي حلقة `for`، و نحوها كنحوها في المحارف:

```
for cheese in cheeses:
print cheese
```

قد يكون هذا كافيا إن كنت تريد قراءة عناصر القائمة فقط. أما لتحديث و كتابة العناصر، فستحتاج إلى المؤشرات، و

الطريقة الشائعة لعمل هذا هي دمج range و len:

```
for i in range(len(numbers)):
     numbers[i] = numbers[i] * 2
```

تمر هذه الحلقة في القائمة و تحدث كل عنصر فيها. ترجع len عدد عناصر القائمة، range ترجع قائمة بالمؤشرات بدءا من المؤشر 0 إلى المؤشر n-1، حيث n هي طول القائمة. و في كل دورة في الحلقة، تأخذ i قيمة المؤشر للعنصر التالي، و عبارة التعيين في متن الحلقة تستخدم i لقراءة اخر قيمة للعنصر و تعين القيمة الجديدة.

حلقة for التي تُستخدم على قائمة فارغة لا تُنفِّذ متن الحلقة أبدا:

```
for x in []:
     print 'This never happens'
```

على الرغم من أنه للقائمة أن تحتوي على قائمة أخرى، إلا أن القائمة المحتواة تعتبر عنصرا واحدا. طول هذه القائمة مثلا هو 4:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 العمليات على القوائم

المؤثر + يضم قائمتين:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

و على نفس النمط، المؤثر * يكرر القائمة عدد المرات المعطاة:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

المثال الاول يكرر [0] أربعة مرات و الثاني يكرر القائمة [1 , 2, 3] ثلاثة.

10.5 شرائح القائمة

مؤثر التشريح (:) يعمل أيضا على القوائم :

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

ان أسقطت المؤشر الاول فإن الشريحة ستبدأ عند البداية، و إن أسقطت الثاني ستستمر إلى نهاية القائمة:

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

و بما أن القوائم تتبدل، فمن المفيد نسخ القائمة قبل القيام بأية عمليات عليها قد تعيد ترتيبها.

إن كان مؤثر التشريح على يسار التعيين سيمكنك من تحديث أكثر من عنصر:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 طرق القوائم

يوفر بايثون طرقا تعمل على القوائم، مثلا `append` تضيف عنصرا جديدا للقائمة:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

و `extend` تأخذ قائمة كقرينة و تضيف كافة العناصر:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

هذا المثال أبقى `t2` بدون تعديل.

و `sort` ترتب عناصر القائمة من الاسفل إلى الاعلى:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

طرق القوائم كلها عقيمة، فهي تعدل القائمة لكنها ترجع `None`، إن كتبت بالخطأ `t = t.sort()` فستخيب النتيجة امالك.

10.7 خطُّ الخرائط ، الفلترة و الاختزال

لتجمع كل أعداد قائمة بإمكانك كتابة حلقة كهذه:

```
def add_all(t):
   total = 0
   for x in t:
      total += x
   return total
```

استُهلت `total` بـ 0، في كل دورة في الحلقة تأخذ `x` عنصرا من القائمة، و الرمز += يوفر طريقا مختصرا لتحديث المتغير عبارة التعيين المزيدة هذه:

```
   total += x
```

مقابلة لهذه:

```
   total = total + x
```

و بينما يتم تنفيذ الحلقة، تُراكم `total` مجموع العناصر، متغير يُستخدم بهذا الشكل يدعى أحيانا مراكم **accumulator**.

و تجميع عناصر القوائم شيء شائع لدرجة أن بايثون يوفر اقترانا جاهزا له، sum :

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

و عملية من هذا النوع تجمع تسلسلا من العناصر في قيمة واحدة تسمى أحيانا اختزال **reduce**.

تمرين 10.1 اكتب اقترانا اسمه nested_sum يأخذ قائمة عشية من الاعداد الصحيحة و يجمع العناصر من كافة القوائم العشية فيها.

قد تود أحيانا المرور على عناصر قائمة بينما تبني أخرى. مثلا، الاقتران التالي يأخذ قائمة من المحارف ثم يرجع قائمة جديدة حروفها كبيرة:

```
def capitalize_all(t):
   res = []
   for s in t:
      res.append(scapitalize())
   return res
```

استُهلت res بقائمة فارغة، و في كل دورة في الحلقة نضيف العنصر التالي. فـ res إذن هي نوع آخر من المراكمات.

عملية كـ capitalize_all تدعى أحيانا **map** لأنها "تخطُّ من" اقتران (و هو في هذه الحالة الطريقة capitalize) إلى كل عنصر في تسلسل.

تمرين 10.2 استخدم capitalize_all لكتابة اقتران اسمه capitalize_nested. يأخذ قائمة عشية من المحارف و يرجع قائمة عشية جديدة كل الحروف بها كبيرة.

عملية شائعة اخرى هي أن تأخذ بعض عناصر القائمة و تكون منها قائمة فرعية. فمثلا الاقتران التالي يأخذ قائمة من المحارف و يرجع قائمة تحتوي على الحروف الكبيرة فقط:

```
def only_upper(t):
   res = []
   for s in t:
      if s.isupper():
         res.append(s)
   return res
```

isupper من طرق المحارف. و ترجع True ان كانت السلسلة تحتوي على حروف كبيرة فقط.

عملية مثل only_upper تسمى فلترة، لأنها تختار بعض العناصر و تهمل الاخرى.

يمكن التعبير عن معظم عمليات القوائم المعروفة بتركيبة من الفلترة و الاختزال و مد الخطوط. و لشيوعها يقدم لنا بايثون خصائص لغوية لدعمها، و من ضمنها الاقتران **map** و رمز يسمى "إنشاء القوائم".

تمرين 10.3 اكتب اقتران يأخذ قائمة من الارقام و يرجع جمعا تراكميا، أي أنه يرجع قائمة جديدة حيث العنصر رقم *i* يكون مجموع أول عنصرين *i*+1 من القائمة الاصلية مثلا الجمع التراكمي لـ [1, 2, 3] هو [1, 3, 6].

10.8 حذف العناصر

هنالك عدة وسائل لحذف عناصر من القائمة، فإن كنت تعرف مؤشر العنصر فبإمكانك استخدام pop:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

تعدل pop القائمة و ترجع العنصر الذي حُذف، و ان لم تزودها بالمؤشر ستحذف و ترجع آخر عنصر في القائمة.

و ان كنت لا تحتاج للعنصر المحذوف فبإمكانك استخدام del :

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

و ان كنت تعلم أي عنصر تريد حذفه لكن ليس المؤشر فبإمكانك استخدام remove :

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

القيمة المرجعة من remove هي None.

و لحذف أكثر من عنصر يمكنك استخدام del مع مؤشر للشريحة المراد حذفها:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

و كالعادة، فالشريحة تختار جميع القيم حتى، و لكن ليس مع، المؤشر الثاني.

تمرين 10.4 اكتب اقترانا اسمه middle يأخذ قائمة و يرجع قائمة جديدة تحتوي على كل عناصر الاولى ما عدا الاول و الاخير فمثلا:

```
middle([1, 2, 3, 4])
```

ترجع:

```
[2, 3]
```

اكتب اقترانا اسمه chop يأخذ قائمة و يعدلها بحذف أول و أخر عنصر فيها و يرجع None.

10.9 القوائم و المحارف

المحارف تسلسل من الحروف و القائمة تسلسل من القيم، لكن قائمة الحروف و المحارف ليستا نفس الشيء. للتحويل من محارف إلى قائمة من الحروف بإمكانك استخدام list :

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

و لأن list هي اسم لاقتران جاهز، فيتوجب عليك تجنب استعمالها كاسم لمتغير، و أنا أيضا اتجنب استخدام l لأنها

تشبه 1 ، و لهذا استخدم t.

الاقتران `list` يقسم المحارف إلى حروف منفردة، ان اردت تقسيم المحارف إلى كلمات فبإمكانك استخدام `split` :

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
```

و لهذه الطريقة قرينة اختيارية اسمها delimiter (فاصل). تحدد أيٍّ من الحروف تريد استعمالها كحدود خارجية للكلمة. المثال التالي يستخدم الشرطة كـ delimiter :

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` هي المقابل لـ `split` . تأخذ قائمة من المحارف و تضيف عناصرها لبعض , و بما أن `join` هي طريقة من طرق المحارف فيجب استدعاؤها على `delimiter` ثم تمرر القائمة كبرمتر:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

كان الفاصل في هذه الحالة هو حرف مسافة، فـ `join` ستضع فراغات بين الكلمات. و لتضيف المحارف بدون مسافات استخدم محارف فارغة '' كفاصل.

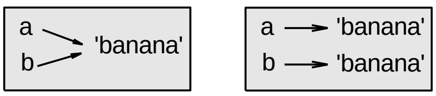
10.10 الكائنات و القيم

ان نفذنا عبارات التعيين هذه:

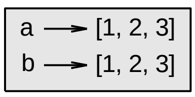
```
a = 'banana'
b = 'banana'
```

فنحن نعرف ان كل من `a` و `b` تشير إلى كائن محارف `'banana'` ، لكننا لا نعرف إن كانتا نفس المحارف. هناك حالتان ، يبينهما الشكل 10.2.

في الحالة الاولى تشير كل من `a` و `b` إلى كائنين مختلفين، لكن لهما نفس القيمة. و في الحالة الثانية تشيران إلى نفس الكائن.



الشكل 10.2: رسم الحالة.



الشكل 10.3 رسم الحالة.

لفحص ما اذا كان متغيران يشيران إلى نفس الكائن، استخدم المؤثر `is` :

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

في هذا المثال أوجد بايثون كائن محارف واحد و أشار كل من a و b إليه، لكن عندما تنشئ قائمتين فستحصل على كائنين:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

و سيبدو رسم الحالة كما في الشكل 10.3.

في هذه الحالة نقول بأن القائمتين متساويتين، لأن لديهما نفس العناصر، الا أنهما ليستا طبق الاصل لأنهما ليستا نفس الكائن. ان تطابق كائنان فسيكونا متساويين، و العكس ليس بالضرورة صحيح.

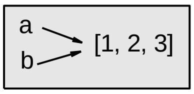
نحن حتى الان نستعمل كلمة "قيمة" مكان "كائن" و العكس، الا أن الأدق هو القول أن للكائن قيمة. فإن نفذت [1, 2, 3] ستحصل على كائن قائمة قيمته تسلسل من الاعداد الصحيحة. و إن كان لسلسلة اخرى نفس العناصر نقول بأن لها نفس القيمة و لا نقول نفس الكائن.

10.11 تعدد المرجعية

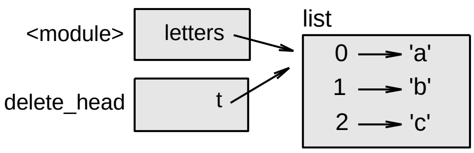
إن كانت a تشير إلى كائن ما ثم عينت a إلى b سيكون كل من المتغيرين مرجع الكائن نفسه:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

رسم الحالة 10.4 يبين هذا الوضع.



الشكل 10.4: الرسم المستف



الشكل 10.5: الرسم المستف

ربط الكائن بمتغير يسمى مرجعية reference. في هذا المثال هناك مرجعيتان لنفس الكائن.

الكائن الذي له أكثر من مرجعية يكون له أكثر من اسم، فنقول أن الكائن متعدد المرجعيات aliased.

و إن كان الكائن متعدد المرجعية غير ثبيت (يعدَّل) ، فأي تغيير تحدثه مرجعية يؤثر في الاخرى:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

رغم الفوائد التي قد تجنى من هذا السلوك الا أنه مخبأ للأخطاء. و في العموم من الآمن تجنب تعدد المراجع، عندما تعمل مع كائنات متبدلة.

تعدد المراجع لا يسبب المشاكل عند العمل على الكائنات الغير متبدلة، في هذا المثال:

```
a = 'banana'
```

```
b = 'banana'
```

لن يغير شيئا كون a و b مرجعان لنفس الكائن أم لا

10.12 القوائم كقرائن

عندما تمرر قائمة لاقتران، يحصل الاقتران على مرجع للقائمة، و إن عدل الاقتران برمتر في القائمة فإن المنادي سيرى النتيجة فمثلا، delete_head تحذف العنصر الاول من القائمة:

```
def delete_head(t):
   del t[0]
```

و هكذا تستعمل:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

البرمتر t و المتغير letters كلاهما مرجع لنفس الكائن و الرسم المستف كما في الشكل 10.5.

من المهم التفريق بين العمليات التي تعدل القوائم و تلك التي تنشئها، فمثلا، طريقة append تعدل القائمة لكن المؤثر + ينشئها:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [4]
>>> print t3
[1, 2, 3, 4]
```

هذا الفرق مهم عندما تكتب اقترانات المفترض فيها تعديل القوائم، فمثلا الاقتران التالي لا يحذف رأس القائمة:

```
def bad_delete_head(t):
   t = t[1:] # WRONG!
```

مؤثر التشريح ينشئ قائمة جديدة و التعيين يجعل t مرجعا لها، لكن ليس لأيٍّ من هذا تأثير على القائمة التي مررت كقرينة.

البديل لهذا هو كتابة اقتران ينشئ و يرجع قائمة جديدة، فمثلا tail ترجع العناصر ما الاول في قائمة ما:

```
def tail(t):
   return t[1:]
```

يترك هذا الاقتران القائمة الاصلية كما هي، و هكذا يستعمل:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

10.13 علاج الاخطاء

عدم الانتباه عند استعمال القوائم (و غيرها من الكائنات المتبدلة) يؤدي عادة إلى انفاق الساعات في البحث عن اسباب الأخطاء. هنا بعض الأخطاء الشائعة و طرق تجنبها:

1. لا تنس بأن غالبية طرق القوائم تعدل القرائن و ترجع None و هذا عكس طرق المحارف، التي ترجع محارف جديدة و تترك الاصلية كما هي.

و ان اعتدت كتابة نصوص برمجية للمحارف هكذا:

```
word = word.strip()
```

فستغريك كتابة نصوص للقوائم هكذا:

```
t = t.sort() # WRONG!
```

لأن sort ترجع None ، فإن العملية التالية التي ستقوم بها مع t ستفشل على الاغلب.

قبل البدء باستعمال طرق و رموز القوائم ، اقرأ وثائقها بتمعن ثم اختبر عملها في الوضع التفاعلي. الوثائق التي تشرح الطرق و الرموز التي تتشارك بها القوائم و التسلسلات الاخرى (كالمحارف) موجودة على:

.http://docs.python.org/2/library/stdtypes.html#typesseq

و الوثائق التي تشرح الرموز و الطرق التي تختص بها التسلسلات المتبدلة موجودة على:

.http://docs.python.org/2/library/stdtypes.html#typesseq-mutable

2. اختر لغة و ابق معها.

من مشاكل القوائم هي كثرة الافكار التي تمكنك من القيام بالاشياء. فمثلا، لحذف عنصر من القائمة يمكنك استخدام pop أو remove أو del و حتى تعيين slice.

و لإضافة عنصر يمكنك استخدام append أو الرمز + . و افتراضا أن t قائمة و x عنصر فيها فالتالي صحيح:

```
t.append(x)
t = t + [x]
```

و التالي خطأ:

```
t.append([x])        # WRONG!
t = t.append(x)      # WRONG!
t + [x]              # WRONG!
t = t + x            # WRONG!
```

جرب كل من هذه الامثلة في الوضع التفاعلي لتتأكد من أنك استوعبت ما تفعله. لاحظ بأن الاخير فقط سيسبب خطأ أثناء التشغيل، أما الباقي فلن ترسل خطأ لكنها لن تفعل المقصود منها.

3. اعمل نسخا لتجنب تعدد المراجع:

ان اردت استخدام طرق كـ sort و التي تعدل على القرائن، لكنك بحاجة لأن تحتفظ بالقائمة الاصلية أيضا، يمكنك عمل نسخة:

```
orig = t[:]
t.sort()
```

في هذا المثال كان يمكنك استخدام الاقتران الجاهز sorted و الذي يرجع قائمة جديدة و مرتبة و يترك الاصلية في حالها الا أنه في هذه الحالة عليك تجنب استخدام sorted كاسم لمتغير.

10.14 المعاني

قائمة list: تسلسل من القيم.

عنصر element: أحد القيم في قائمة (أو أي تسلسل اخر)، يسمى أيضا item .

مؤشر index: قيمة عدد صحيح تؤشر إلى عنصر في القائمة.

المرور على القائمة list traversal: الوصول إلى كل عنصر في القائمة بالتوالي.

خط الخرائط Mapping: علاقة يرتبط بها كل عنصر في مجموعة مع عنصر في مجموعة أخرى. فمثلا القائمة هي ارتباط بين المؤشرات و العناصر.

المراكم accumulator: متغير يستخدم في حلقة لتجميع (أو مراكمة) النتيجة.

التعيين المزيد augmented assignment: عبارة تُحدِّث قيمة متغير باستخدام رمز حوسبي كـ =+.

اختزال reduce: نمط عملياتي يتم فيه المرور على عناصر تسلسل ثم مراكمتها في نتيجة وحيدة.

خط Map: نمط عملياتي يتم فيه المرور على عناصر تسلسل و القيام بعملية ما على كل منها.

الفلترة filter: نمط عملياتي يتم فيه المرور على عناصر قائمة و انتقاء ما يفي بشروط ما منها.

كائن object: شيء يمكن لمتغير أن يكون مرجعا له و للكائن نمط و قيمة.

مساوي equivalent: أن تكون لها نفس القيمة.

طبق الاصل identical: أن يكون نفس الكائن (و بالتالي مساوي).

مرجع reference: الرابطة بين متغير و قيمة.

تعدد المرجعية Aliasing : ظرف يكون فيه متغيرين أو كثر مرجعا لنفس الكائن.

المحدد delimiter : حرف أو محارف تستخدم لبيان الموقع الذي سيتم تقسيم المحارف عنده.

10.15 تمارين

تمرين 10.6: اكتب اقترانا اسمه `is_sorted` يأخذ قائمة كبرمتر و يرجع `True` ان كانت عناصرها مرتبة تنازليا، و `False` ان كانت غير ذلك. بامكانك الافتراض (كشرط مسبق) أنه بالامكان مقارنة عناصر القائمة بمؤثرات النسبة <, > , الخ. مثلا:

```
is_sorted([1, 2, 3])
```

سترجع `True` أما:

```
is_sorted([ 'b', 'a'])
```

سترجع `False`.

تمرين 10.7 ان كانت لكلمتين نفس الحروف لكن مختلفة الترتيب يقال لهما anagram (جناس ناقص؟). اكتب اقترانا و سمه `is_anagram` يأخذ محارفتين و يرجع `True` إن كانتا جناسا ناقصا.

تمرين 10.8 ما يسمى مفارقة تاريخ الميلاد:

1. اكتب اقترانا اسمه `has_douplicates` يأخذ قائمة و يرجع `True` ان ظهر أي عنصر أكثر من مرة و يجب ألا تعدل على الاصل
2. ان كان هناك 23 طالبا في صفك، ما احتمال أن يكون اثنان منكما بنفس تاريخ الميلاد؟ بامكان توقع هذا الاحتمال بإنشاء عينة عشوائية لـ 23 تاريخ ميلاد و فحص المتساوية منها تلميح: يمكنك انشاء تواريخ ميلاد عشوائية باستخدام اقتران `randint` في مديول `random`

بامكانك القراءة عن هذه المسألة على:

.http://en.wikipedia.org/wiki/Birthday_paradox

و يمكنك تحميل حلي للمسألة من .http://thinkpython.com/code/birthday.py

تمرين 10.9 اكتب اقترانا اسمه `remove_duplicates` يأخذ قائمة و يرجع قائمة جديدة بها العناصر الفريدة فقط من القائمة الاصلية. تلميح: ليس من الضروري أن تكون بنفس الترتيب

تمرين 10.10 اكتب اقترانا يقرأ الملف `words.txt` ثم ينشئ قائمة تكون فيها كل كلمة عنصرا اكتب نسختين من هذا الاقتران، واحدة تستعمل `append` و الثانية تستعمل المزيدة `t = t + [1]` ، ايها تأخذ وقتا أطول للتنفيذ؟ لماذا؟

تلميح: استخدم مديول `time` لقياس استهلاك الوقت. الحل:

.http://thinkpython.com/code/wordlist.py

تمرين 10.11 لفحص ما إذا كانت كلمة موجودة في قائمة الكلمات قد تستعمل المؤثر `in` ، الا أنه يكون بطيئا لبحثه في كل الكلمات بالترتيب.

و لكون الكلمات مرتبة أبجديا، سنتمكن من تسريع العملية بالبحث التنصيفي (يعرف أيضا بالبحث الثنائي "بايناري") و هو يشبه ما تقوم به عند البحث عن كلمة في المعجم.

ستبدأ في المنتصف و ترى إذا ما كانت كلمة البحث تسبق كلمة المنتصف هذه أو تلحقها، فإن كانت تسبقها فستبدأ البحث في منتصف النصف الاول و ترى أين تقع كلمة البحث منه، و هكذا.

و ايا كان الحال فستختصر نصف البحث في كل مرة. إن كان في قائمة كلمات 113809 كلمات، سيتطلب الامر 17 خطوة فقط لإيجاد كلمة البحث أو القول بعدم وجودها في القائمة.

اكتب اقترانا و سمه `bisect` يأخذ قائمة مرتبة و قيمة مستهدفة ثم يرجع المؤشر لهذه القيمة في القائمة، أو يرجع `None` في حالة عدم وجودها.

أو يمكنك قراءة وثائق الاقتران الجاهز `bisect` و استعماله!

الحل: .http://thinkpython.com/code/inlist.py

تمرين 10.12 تكون الكلمتان منعكستان إن كانت احداهما معكوس حروف الاخرى. اكتب برنامجا يجد كل الازواج المنعكسة في قائمة الكلمات.

الحل: .http://thinkpython.com/code/reverse_pair.py

تمرين 10.13 الكلمتان تتشابكان إن أخذنا حرفا من كل منهما في كل مرة لنكون في النهاية كلمة جديدة مثلا: *shoe* و **cold** تكونان ***schooled*** الحل: .http://thinkpython.com/code/interlock.py

عرفان: استلهم هذا التمرين من http://puzzlers.org.

1- اكتب برنامجا يجد كل ازواج الكلمات المتشابكة في قائمة الكلمات. تلميح: لا تسرد جميع الازواج.
2- هل بإمكانك ايجاد الكلمات المتشابكة ثلاثيا، اي أن كل ثالث حرف هو ما يشكل الكلمة الجديدة؟

الفصل الحادي عشر

القواميس

القاموس كالقائمة، الا أنه أعَم. في القائمة تكون المؤشرات أعداد صحيحة، في القاموس فيمكن لها أن تكون من أي نمط، تقريبا.

يمكنك الفكير بالقاموس كارتباط بين مجموعة من المؤشرات (و التي تسمى **مفاتيح**) و مجموعة من القيم. فكل مفتاح يرتبط بقيمة، هذا التشارك بين المفاتيح و القيم يدعى زوجي المفتاح-القيمة key-value pair و أحيانا عنصر item.

و كمثال سننشئ قاموسا يربط بين الانجليزية و الاسبانية، فتكون القيم و المفاتيح جميعها محارف.

الاقتران dict ينشئ قاموسا جديدا بدون عناصر. و لأن dict هي اسم لاقتران جاهز، فإن عليك تجنب استخدامها كاسم لمتغير.

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

تمثل الاقواس المتموجة {} قاموسا فارغا، و لتضيف عناصر للقاموس ستستخدم الاقواس المربعة:

```
>>> eng2sp['one'] = 'uno'
```

يخلق هذا السطر عنصرا يربط بين المفتاح one و القيمة uno، و إن طبعنا القاموس مرة اخرى سنرى زوجي المفتاح-القيمة و بينهما نقطتان:

```
>>> print eng2sp
{'one': 'uno'}
```

صيغة المخرجات هي أيضا صيغة الادخال فيمكنك مثلا انشاء قاموس جديد به ثلاثة عناصر:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

الا انك ستتفاجأ عند طباعة enf2sp :

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

فترتيب أزواج القيم-المفاتيح ليس كما كان مُدخَل، بل حتى إنك إن طبعت نفس المثال على حاسوبك ستحصل على نتيجة مختلفة، الواقع هو أن ترتيب العناصر في القاموس غيبيا.

الا أنها ليست مشكلة، لأن العناصر في القاموس ليست مؤشرة بأعداد صحيحة، فبدلا من ذلك ستستعمل المفاتيح للبحث عن القيم المقابلة لها:

```
>>> print eng2sp['two']
'dos'
```

المفتاح two يرتبط دائما بالقيمة dos، اذن فترتيب العناصر لا يهم.

ان كان المفتاح ليس في القاموس فستحصل على استثناء:

```
>>> print eng2sp['four']
KeyError: 'four'
```

يعمل الاقتران `len` على القواميس فهو يرجع عدد أزواج المفاتيح-القيم

```
>>> len(eng2sp)
3
```

و المؤثر `in` يعمل أيضا على القواميس، فهو يؤكد ان كان شيئا يظهر كمفتاح في القاموس (أن يظهر كقيمة ليس جيدا كفاية).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

لترى إذا ما كان شيء يظهر كقيمة في القاموس، يمكنك استخدام الطريقة `values`، و التي ترجع القيم على شكل قائمة، ثم بعد ذلك استعمل المؤثر `in`:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

في القواميس يَستخدم المؤثر `in` لوغرتمية مختلفة عن تلك التي يستخدمها في القوائم، في القوائم يستخدم لوغرتمية بحث search ، كما في القسم 8.6، لذلك عندما تتضخم القائمة يطول وقت البحث في تناسب مباشر. في القواميس، يستخدم بايثون لوغرتمية تدعى جدول تقطيع hashtable لها خاصية مميزة، و هي أنه مهما بلغ عدد عناصر القاموس فإن المؤثر `in` سيأخذ نفس الوقت تقريبا، لن اشرح كيف يمكن ذلك، لكن يمكنك القراءة عنه هنا:

.http://en.wikipedia.org/wiki/Hash_table

تمرين 11.1 اكتب اقترانا يقرأ كل كلمات `words.txt` و يخزنها كمفاتيح في قاموس، لا يهم ما هي القيم، ثم يمكنك بعدها استخدام الرمز `in` كوسيلة سريعة لفحص ما إذا كانت محارف ما موجودة في القاموس.

ان قمت بالتمرين 10.11 فبإمكانك مقارنة سرعة العملية هنا مع السرعة في تطبيقها على القوائم.

11.1 القواميس كمجموعة من العدادات

افرض أن لديك محارف و طلب منك معرفة عدد مرات ظهور كل حرف فيها، هنالك عدة وسائل لحلها:

1- ستوجد 26 متغيرا، واحدا لكل حرف في الابجدية، ثم تمر بعناصر السلسلة، ثم لكل حرف تمر به ستزيد قيمة العداد المقابل له، و قد تستخدم مشروطة عشية.

2- قد توجد قائمة بـ 26 عنصر، ثم ستحول كل حرف إلى رقم (باستخدام الاقتران الجاهز `ord`)، ثم ستستخدم الرقم كمؤشر في القائمة، ثم تزيد العداد المقابل.

3- قد تنشئ قاموسا تكون الحروف فيه المفاتيح و العدادات كالقيم المقابلة، و عندما تصادف الحرف لأول مرة ستضيف عنصرا إلى القاموس، و بعد ذلك تزيد قيمة العنصر الموجود.

كل من هذه الخيارات يقوم بنفس العملية الحوسبية، الا أن كل منها يطبقها بشكل مختلف.

التطبيق هو طريقة للقيام بالعمليات الحوسبية implementation. بعض التطبيقات أفضل من الاخرى، فمثلا من محاسن استخدام القواميس أنه لا يتطلب منا معرفة مسبقة بأي من الحروف سيظهر في المحارف، فقط علينا إيجاد مكان للحروف التي ستظهر.

النص البرمجي له سيكون كهذا:

```
def histogram(s):
   d = dict()
```

```
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

اسم الاقتران كان histogram المدرج التكراري، و هو مصطلح احصائي لمجموعة من العدادات (أو الترددات).

ينشئ السطر الاول في الاقتران قاموسا فارغا، و تمر حلقة for في المحارف، و في كل دورة، إن لم يكن الحرف c (اسم المتغير) موجودا في القاموس سيخلق عنصرا جديدا مفتاحه c و قيمته الابتدائية 1 (بما أننا قد رأينا هذا الحرف مرة واحدة) و إن كان الحرف c موجودا في القاموس سنزيد d[c].

هكذا يعمل البرنامج:

```
>>> h = histogram('brontosaurus')
>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

المدرج التكراري يبين أن الحروف a و b ظهرت مرتين، و هكذا.

تمرين 11.2 للقواميس طريقة اسمها get تأخذ مفتاحا و قيمة افتراضية ان وجد المفتاح في القاموس فإن get سترجع القيمة المقابلة له، و الا سترجع القيمة الافتراضية مثلا:

```
>>> h = histogram('a')
>>> print h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

استخدم get لكتابة نسخة موجزة من histogram يجب أن تتمكن من الاستغناء عن عبارة if.

11.2 التدوير و القواميس

ان استخدمت قاموسا في حلقة for فإنها ستمر على مفاتيح القاموس. مثلا print_hist تطبع كل مفتاح و القيمة المقابلة له:

```
def print_hist(h):
    for c in h:
        print c, h[c]
```

مخرجات هذا الاقتران هي :

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

و مرة أخرى، ليس للمفاتيح ترتيب محدد.

تمرين 11.3 للقواميس طريقة اسمها keys ترجع مفاتيح القاموس، و بدون ترتيب محدد، على شكل قائمة.

عدل print_hist ليطبع المفاتيح و القيم المقابلة لكن بالترتيب الابجدي.

11.3 البحث العكسي

ان اعطيت قاموسا اسمه d و مفتاح k سيكون من السهل عليك ايجاد القيمة المقابلة v = d[k]، هذه العملية تدعى البحث lookup.

أما ان اعطيت القيمة v و طلب ايجاد المفتاح k؟ فلديك مشكلتان: الاولى أنه قد يكون هناك اكثر مفتاح يرتبط بالقيمة v، و حسب التطبيق، فقد تتمكن من التقاط قيمة واحدة، أو قد تضطر لانشاء قائمة تحتوي جميع القيم. الثانية، عدم وجود نحو بسيط للقيام بالبحث العكسي، فعليك البحث.

هذا اقتران يأخذ قيمة و يرجع أول مفتاح يقترن بالقيمة:

```
def reverse_lookup(d, v):
   for k in d:
      if d[k] == v:
         return k
raise ValueError
```

هذا الاقتران ليس سوى مثال اخر على نمط البحث، إلّا انها تستخدم مزية لم نرها من قبل، raise. تتسبب عبارة raise باستثناء، هنا تسببت بـ ValueError و الذي يعني في الغالب وجود خطب ما في قيمة البرمتر.

إن وصلنا إلى نهاية الحلقة فسيعني أن v لا تظهر في القاموس كقيمة، و عليه فسنرفع (raise) استثناءا.

هاك مثال على بحث عكسي ناجح:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> print k
r
```

و بحث فاشل:

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "<stdin>", line 5, in reverse_lookup
ValueError
```

أن ترفع انت استثناءا أو أن يرفعه بايثون فالنتيجة واحدة: سيطبع ملاحقة و رسالة خطأ.

تأخذ عبارة raise تفاصيل الخطأ كقرينة اختيارية، فمثلا:

```
>>> raise ValueError('value does not appear in the dictionary')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ValueError: value does not appear in the dictionary
```

البحث العكسي ابطأ من البحث الامامي، لذلك سيعاني أداء حاسوبك ان قمت به بكثرة أو كان حجم القاموس كبيرا.

تمرين 11.4 عدل على reverse_lookup بحيث يبني و يرجع قائمة بكل المفاتيح التي تقترن بـ v أو قائمة فارغة ان لم توجد مفاتيح مرتبطة.

11.4 القواميس و القوائم

يمكن للقوائم أن تظهر كقيم في القاموس، فمثلا ان كان لديك قاموسا يربط الحروف بالترددات، فقد تود أن تعكسه، أي خلق قاموس يربط الترددات بالحروف، و بما أنه من الممكن وجود عدة حروف لنفس التردد، فستكون كل قيمة في القاموس المعكوس قائمة.

هاك اقترانا يعكس قاموس:

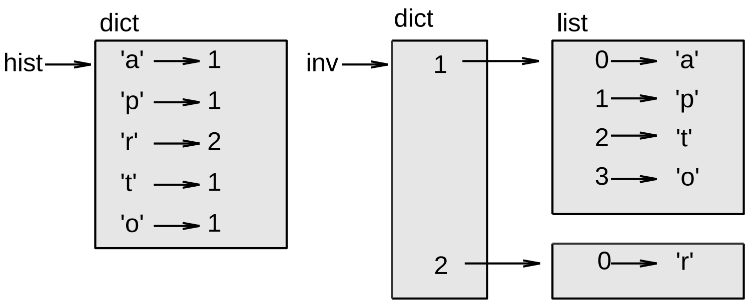
```
def invert_dict(d):
   inverse = dict()
   for key in d:
      val = d[key]
      if val not in inverse:
         inverse[val] = [key]
      else:
         inverse[val]append(key)

   return inverse
```

في كل دورة في الحلقة تأخذ key مفتاحا من d و تأخذ val القيمة المقابلة. و إن لم تكن val موجودة في inverse سيعنى أننا لم نرها من قبل، فننشئ عنصرا و نستهله بـ singleton (قائمة تحتوى على عنصر وحيد). و إلا فقد رأينا هذه القيمة من قبل، فنضيف المفتاح المقابل إلى القائمة.

هذا مثال:

```
>>> hist = histogram('parrot')
>>> print hist
```



الشكل 11.1 رسم الحالة

```
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> print inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

الشكل 11.1 هو رسم حالة يبين hist و inverse. القاموس ممثّل بصندوق يكون النمط dict فوقه، و زوجي المفتاح-القيمة بداخله. ان كانت القيم اعداد صحيحة أو قيم عائمة أو محارف فأنا ارسمها في العادة داخل الصندوق، لكنني في العادة ارسم القوائم خارجه، فقط لابقاء الرسم بسيطا.

يمكن للقوائم أن تكون قيما في قاموس، كما يبين هذا المثال، لكن لا يمكن لها أن تكون مفاتيح. و هذا ما سيحدث ان حاولت:

```
>>> t = [1, 2, 3]
```

```
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

ذكرت في السابق بأن القواميس مطبقة باستخدام جدول تقطيعي، مما يعني أن المفاتيح يجب ان تكون hashable تقطيعية.

الهاش hash هو اقتران يأخذ قيمة من أي نوع و يرجع عددا صحيحا. تستخدم القواميس هذه الاعداد، و تسمى قيم التقطيع، لتخزين ازواج المفاتيح-القيم و البحث فيها.

سيعمل هذا النظام جيدا لو كانت المفاتيح غير متبدلة. لكن لو كانت متبدلة ، كالقوائم، ستحدث أمور غير محمودة، فمثلا عندما تنشئ أزواج مفاتيح-قيم فإن بايثون يهيش المفتاح و يخزنه في الموقع المقابل، و إن عدلت المفتاح و هيشتَه مرة اخرى، سيذهب إلى موقع اخر. في هذه الحالة سيكون هناك مدخلان لنفس المفتاح، أو قد لا تتمكن من العثور على المفتاح. و في كلتا الحالتين لن يعمل القاموس بشكل صحيح.

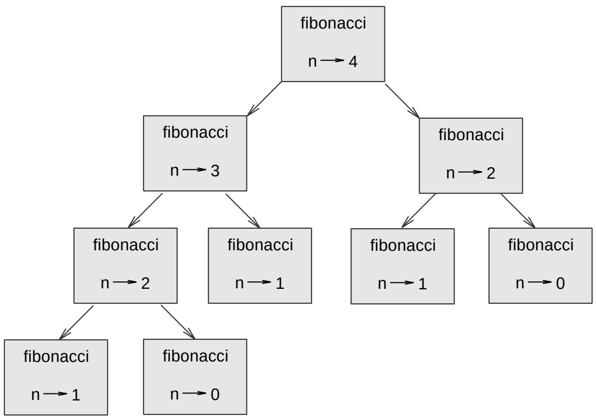
لهذا السبب كان على المفاتيح و ليس على الأنماط المتبدلة أن تكون تقطيعية. أبسط طريقة للتحايل على هذا التقييد هو استعمال الصفوف tuples، و هو ما سنراه في الفصل التالي.

بما أن القوائم و القواميس متبدلة، فلا يمكن استخدامها كمفاتيح لكن يمكن استخدامها كقيم.

تمرين 11.5 اقرأ وثائق طريقة القاموس `set.default` و استخدمها لكتابة نسخة موجزة من `invert_dict` الحل: `http://thinkpython.com/code/invert_dict.py`.

11.5 المذكرات

ان كنت قد لعبت قليلا بفيبوناشي من القسم 6.7 فقد تكون قد لاحظت أنه كلما كبرت القرينة التي تزودها، كلما طال الوقت ليبدأ الاقتران بالعمل، و أكثر من ذلك، فقد ازداد زمن التشغيل أيضا.



الشكل 11.2: رسم النداء

لتفهم السبب،انظر إلى الشكل 11.2 الذي يبين رسم النداء لـ `fabonacci` مع `n=4` .

يبين رسم النداء مجموعة من اطارات الاقترانات مع خطوط تصل كل اطار إلى الإطار الذي ينادي عليه. في رأس الرسم كانت `fibonacci` مع `n=4` تنادي `fibonacci` مع `n=3` و مع `n=2`. و بدورها، فإن `fabonacci` مع `n=3` تنادي `fibonacci` مع `n=2` و `n=1` و هكذا.

ان عددت كم مرة نودي على `fibonacci(0)` و , `fibonacci(1)` سترى بأن هذا الحل ليس كفؤا للمسألة، و

سيزداد الوضع سوءا كلما كبرت القرينة.

متابعة القيم التي استعملت و تخزينها في قاموس هو أحد الحلول. فالقيمة التي خزنت لاستعمال لاحق تسمى مذكِّرة memo

هذه نسخة متَذكِّرة لـ fabonacci:

```
known = {0:0, 1:1}
   def fibonacci(n):
   if n in known:
      return known[n]

   res = fibonacci(n-1) + fibonacci(n-2)
   known[n] = res
   return res
```

known هو قاموس يتابع و يحتفظ بأرقام فِبوناشي التي تعرفنا عليها. يبدأ : 0 تصل بـ 0 و 1 تصل بـ 1.

في كل مرة ينادى fabonacci فإنه يتحقق من المذكرة known، فإن كانت النتيجة مدونة هناك سيرجع حالا. و إلا فإن عليه أن يحسب قيمة جديدة، و يضيفها للقاموس، ثم يرجعها.

تمرين 11.6 شغل هذه النسخة من fabonacci و كذلك النسخة الاصلية على نطاق من البرمترات، ثم قارن زمن التشغيل.

تمرين 11.7 حول اقتران أكِرمان في التمرين 6.5 إلى اقتران بمذكرة، و انظر ان كان هذا التحويل يسمح بتقييم الاقتران بقرائن أكبر تلميح: لا.

الحل: http://thinkpython.com/code/ackermann_memo.py.

11.6 المتغيرات العمومية

في المثال السابق وُجِدت known خارج الاقتران، لهذا فهي تنتمي لإطار خاص يسمى الرئيسي __main__. تسمى المتغيرات في __main__ أحيانا عمومية global، لأن الوصول اليها ممكن من اي اقتران. و على خلاف المتغيرات الموضعية التي تختفي بمجرد انتهاء اقترانها من العمل، فالمتغيرات العمومية تستمر في الوجود من نداء لاقتران إلى آخر.

من الشائع استخدام المتغيرات العمومية للرايات flags ، و عبارة عن هي متغيرات بوليان ترفع "علما" ان تحقق الشرط. فمثلا ، تستخدم بعض البرامج راية تسمى verbose للتحكم بمستوى تفاصيل المخرجات:

```
verbose = True

def example1():
   if verbose:
      print 'Running example1'
```

و إن حاولت إعادة تعيين متغير عمومي ستفاجأ، يُفترض في المثال التالي أن يتحقق من إذا ما نودي على الاقتران:

```
def example2():
   been_called = True        # WRONG
```

الا أنك عندما تشغله سترى بأن قيمة been_called لا تتغير. المشكلة هي أن example2 يخلق متغيرا موضعيا اسمه been_called، و المتغير الموضعي يختفي بانتهاء الاقتران، و لا يكون له أثر على المتغير العمومي.

عند التعيين لمتغير عمومي داخل الاقتران، عليك الاعلان عن المتغير قبل استعماله:

```
been_called = False
```

```
def example2():
   global been_called
   been_called = True
```

و كأن عبارة global تقول للمفسّر: "في هذا الاقتران، عندما أقول been_called فإنني أعني المتغير العمومي، لا تخلق متغيرا موضعيا."

هذا مثال يحاول تحديث متغير عمومي:

```
count = 0
def example3():
   count = count + 1 # WRONG
```

و إن شغلته ستحصل على:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

يفترض بايثون أن count موضعيا، مما يعني أنك تقرأه قبل أن تكتبه.

الحل، و مرة اخرى، هو اعلان count متغيرا عموميا:

```
def example3():
   global count
   count += 1
```

ان كان المتغير العمومي غير ثبيت فيمكنك تعديله دون الاعلان عنه:

```
known = {0:0, 1:1}

def example4():
   known[2] = 1
```

إذن فبإمكانك الاضافة و الحذف و استبدال عناصر القائمة العمومية او القاموس العمومي، الا أنه عليك الاعلان عن متغير عمومي إن أردت اعادة تعيينه:

```
def example5():
   global known
   known = dict()
```

11.7 الاعداد الصحيحة الطويلة

ان قمت بحساب fabonacci(50) ستحصل على:

```
>>> fibonacci(50)
12586269025L
```

تشير L في نهاية النتيجة إلى أن العدد الصحيح طويل، أو أن النمط long، في بايثون3 اختفت long ، كل الاعداد الصحيحة، حتى الكبيرة جدا منها لها نوع واحد int.

مجال الاعداد الصحيحة (النمط int) محدود، و هذه الاعداد قد تصبح كبيرة بشكل مبالغ فيه، و منثم ستستهلك الوقت و الحيز.

تعمل المؤثرات الحسابية، و كذلك اقترانات مديول math أيضا، على الاعداد الصحيحة الطويلة، ، لذلك فأي اقتران يعمل مع int سيعمل مع long.

فحينما تكون نتيجة العملية الحوسبية طويلة، سيقوم بايثون بتحويلها إلى عدد صحيح طويل (long) :

```
>>> 1000 * 1000
```

```
1000000
>>> 100000 * 100000
10000000000L
```

كانت النتيجة في الحالة الاولى int و في الثانية long.

تمرين 11.8 تعتبر الاقترانات الاسية للأعداد الصحيحة الطويلة اساس لوغرتميات ار اس ايه للتشفير بواسطة مفتاح عام اقرأ عن لوغرتمية RSA على: http://en.wikipedia.org/wiki/RSA_(algorithm).

ثم اكتب اقترانا يشفر و يحل تشفير الرسائل.

11.8 علاج الاخطاء

كلما كبر حجم مجموعات البيانات، أصبح التعامل مع علاج اخطائها عن طريق طباعتها و فحصها يدويا أصعب. اليك بعض النصائح لعلاج البيانات الضخمة:

صغر حجم المدخلات: إن أمكنك، فصغر حجم المدخلات. مثلا إن كان البرنامج يقرأ ملف نصي فابدأ بأول عشرة سطور، أو بأصغر عينة تجدها، يمكنك مثلا تعديل الملفات نفسها، أو (الافضل) تعديل البرنامج بحيث يقرأ أول n سطور.

ان كان هناك خطأ، يمكنك إنقاص n إلى الحد الذي يظل فيه الخطأ ظاهرا، ثم زدها تدريجيا كلما وجدت خطأ و صححته.

تفقد التلخيصات و الانماط: بدلا من طباعة و فحص مجمل البيانات، جرب طباعة تلخيصات البيانات: مثلا عدد العناصر في قاموس أو مجموع قائمة أرقام.

خطأ شائع عند التشغيل يقع عندما تكون القيمة من نمط غير مناسب، و لاصطياد خطأ كهذا يكفي عادة طباعة نمط القيمة.

اكتب فحصا ذاتيا: بإمكانك احيانا كتابة نص برمجي يتفحص وجود الأخطاء اليا. مثلا إن كنت تكتب حوسبة لإيجاد المعدل لقائمة من الارقام، فيمكنك إضافة نص يتأكد من أن النتيجة ليست أكبر من أكبر عنصر في القائمة أو أنها ليست أصغر من أصغر عنصر فيها، يسمى هذا الفحص بفحص المعقولية (sanity check) لأنه يختبر لك إن كانت النتائج "معقولة".

نوع اخر من الفحوص يقارن نتيجة عمليتين حوسبيتين و يقرر إذا ماكانتا متسقتين و يسمى "فحص التناسق".

الطباعة المهندمة للمخرجات: صياغة شكل مخرجات عملية صيد الأخطاء يسهل اصطيادها. رأينا مثالا في القسم 6.9، فمديول pprint يزودنا باقتران جاهز يظهر الانماط بشكل مقروء (انسانيا).

مرة أخرى، الوقت الذي تستخدمه في تركيب السقّالات سيقلل الوقت الذي تستخدمه في البحث عن أسباب الاخطاء.

11.9 المعاني

قاموس dictionary : تخطيط بين مجموعة من المفاتيح و القيم المقابلة لها.

زوج مفتاح-قيمة key-value pair: تمثيل للتخطيط بين مفتاح و قيمة.

عنصر item: اسم اخر لزوجي مفتاح-قيمة.

مفتاح key: كائن يظهر في القاموس كالقسم الاول من زوجي مفتاح-قيمة.

قيمة value: كائن يظهر في القاموس كالجزء الثاني من زوجي مفتاح-قيمة و عليك اعتبار هذا التعريف اكثر تحديدا من التعريف السابق لكلمة قيمة.

تطبيق implementation: أسلوب للقيام بالعمليات الحوسبية.

جداول التقطيع hashtable : لوغرتمية تستخدمها قواميس بايثون.

اقتران تقطيعي hash function : اقتران تستخدمه جداول التقطيع لحساب موقع المفتاح.

تقطيعي hashable : نمط له وظيفة تقطيعية. الانماط الثبيتة كالاعداد الصحيحة و العائمة و المحارف تقطيعية. و الانماط المتعدلة كالقوائم و القواميس ليست تقطيعية.

بحث lookup: عملية معجمية بأخذ مفتاح و العثور على القيمة المقابلة له.

بحث عكسي reverse lookup: عملية معجمية بأخذ قيمة و البحث عن مفتاح أو أكثر لها.

وحيدة singleton: قائمة (أو اي تسلسل اخر) بها عنصر وحيد.

رسم النداء call graph: رسم يظهر كل اطار خلق خلال تنفيذ البرنامج، يرسم فيه سهما من المنادي إلى المنادى.

مدرج تكراري histogram: مجموعة من العدادات.

مذكرة memo : قيمة حوسبة تُدون في مذكرة لتجنب القيام بالحوسبة ذاتها مستقبلا.

متغيرات عمومية global variables: متغير يعرف خارج الاقتران يمكن الوصول للمتغيرات العمومية من داخل أي اقتران.

راية flag: متغير بوليان يستخدم للإشارة إن كان الشرط True.

إعلان decliration: عبارة كـ global تبلغ المفسِّر شيئا يتعلق بالمتغير.

11.11 تمارين

تمرين 10.9 إن كنت قد عملت على التمرين 108 فسيكون لديك الاقتران `has_duplicates` و الذي يأخذ قائمة كبرمتر و يرجع `True` إن ظهر أي كائن أكثر من مرة فيها.

استخدم قاموسا لكتابة نسخة أسرع و أبسط من `has_duplicates`. الحل: `http://thinkpython.com/code/has_duplicates.py`.

تمرين 11.10 يقال لكلمتين "تدوران" ان كان تدوير احداهما يعطي الاخرى، (أنظر `rotate_word` في التمرين 8.12)

اكتب برنامجا يقرأ قائمة كلمات و يجد كل الازواج التي تدور الحل:

`http://thinkpython.com/code/rotate_pairs.py`.

تمرين 11.11 هذه أحجية أخرى من :Car Talk

"ارسلت هذه من قبل صديق اسمه دان أوليري. فلقد صادف لفظة شائعة من خمسة حروف مؤخرا و كانت لها الخاصية الفريدة التالية إن حذفت حرفها الاول ستجعل الباقي لفظة متجانسة لفظيا من الكلمة الاصلية، أي كلمة تسمع كما تسمع الكلمة الاصلية ثم إن أرجعت الحرف الاول و حذفت الثاني ستجعل الكلمة الناتجة متجانسة لفظيا أيضا مع الاصلية السؤال هو ما هي الكلمة الاصلية ؟

سأعطيك مثالا لا يعمل أنظر إلى الكلمة الخماسية(wrack (W, R, A, C, K و تعني حطام أو خراب ان حذفت الحرف الاول فسأحصل على RACK و تعني رف ، و أرجعت W ثم حذفت R فستحصل على WACK و هي كلمة ذات معنى الا أنها ليست متجانسة لفظيا سابقاتها.

الا أنه هناك بالفعل كلمة واحدة على الاقل، نعلم عنها نحن و دان، إن حذفت أي من حرفيها الاولين تلد كلمة متجانسة لفظيا. السؤال ما هي الكلمة؟"

يمكنك استخدام القاموس من تمارين 11.1 لفحص إذا ما كانت المحارف هذه موجودة في قائمة الكلمات.

للتأكد إذا ما كانت كلمتان متجانستان لفظيا يمكنك الاطلاع على القاموس اللفظي CMU و يمكن تحميله من http://www.speech.cs.cmu.edu/cgi-bin/cmudict.

أو من http://thinkpython.com/code/c06d.

و يمكنك تحميل http://thinkpython.com/code/pronounce.py. و به اقتران اسمه read_dictionary يقرأ القاموس اللفظي و يرجع قاموس بايثون يربط بين كل كلمة و المحارف التي تصف اللفظ الاساسي لها.

اكتب برنامجا يسرد جميع الكلمات التي تصلح حلا لأحجية Car Talk.

الحل: http://thinkpython.com/code/homophone.py.

الفصل الثاني عشر

التوبلات Tuples

12.1 التوبلات ثبيتة

التوبلات هي تسلسل من القيم. و يمكن للقيمة أن تكون من أي نمط، و هذه القيم تكون مؤشَّرة بأعداد صحيحة، فهي شبيهة من هذه الناحية بالقوائم. أهم الفروق بينهما هو أن التوبلات لا تتبدل.

نحويا، فالتوبلات قائمة تفصل بين قيمها الفواصل:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

بالامكان وضع التوبلات بين قوسين، إلا أنه ليس الزاميا:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

لكي تنشئ توبل، عليك كتابة فاصلة في نهاية العبارة:

```
>>> t1 = 'a',
>>> type(t1)
<type 'tuple'>
```

القيمة التي توضع بين قوسين لا تعتبر توبل:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

و طريقة أخرى لإنشاء توبل هي الاقتران الجاهز `tuple`، إن استخدمته بدون قرائن فإنه سينشئ توبل فارغة:

```
>>> t = tuple()
>>> print t
()
```

ان كانت قرينته تسلسلا ما (قائمة، محارف أو توبل)، فستكون النتيجة توبلا من عناصر ذلك التسلسل:

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

و لكون `tuple` اسما لاقتران جاهز، فيجب عليك تجنب استخدامه كاسم لمتغير.

معظم مؤثرات القوائم تعمل على التوبل، فالاقواس المربعة ستشير إلى العنصر:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

و المؤثر `slice` سيحدد نطاقا من عناصر التوبل:

```
>>> print t[1:3]
('b', 'c')
```

الا أنك ان حاولت تعديل أحد عناصر التوبل فستحصل على خطأ:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

لا يمكنك التعديل على عناصر التوبل، إلا أنه يمكنك استبدال توبلا بآخر:

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

12.2 التعيين للتوبل

غالبا ما نستفيد من تبادل قيم متغيرين، و في طرق التعيين التقليدية تضطر لاستعمال متغير مؤقت، فلكي تبادل قيم a و b :

```
>>> temp = a
>>> a = b
>>> b = temp
```

البديل هو من العيار الثقيل، **فتعيينات التوبل** أكثر أناقة:

```
>>> a, b = b, a
```

توبلا من المتغيرات في القسم الايسر و في الايمن توبلا من التعبيرات. كل قيمة تعين إلى المتغير المقابل لها و يتم تقييم جميع القيم التي على الجانب الايمن قبل أي تعيين.

يجب أن يكون عدد المتغيرات على الجانب الايسر مساو لعدد القيم على الايمن:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

في العموم، يمكن للجانب الايمن أن يكون أي نمط من الانماط (محارف، قائمة أو توبل) فمثلا، لكي تفلق عنوان بريد الكتروني إلى اسم مستخدم و نطاق (domain) قد تكتب:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

القيمة المرتجعة من `split` تكون قائمة بها عنصرين، العنصر الأول يعين لـ `uname` و الثاني يعين لـ `domain`.

```
>>> print uname
monty
>>> print domain
pythonorg
```

12.3 التوبل كقيم مرجَعة

حرفيا، للإقتران أن يرجع قيمة واحدة فقط، إلا أنه إن كانت القيمة توبل، فتأثيرها يكون كأرجاع عدة قيم. فمثلا ان اردت تقسيم عددين صحيحين و أرجاع ناتج القسمة و باقيها، فلن يكون عمليا أن تحسب `x/y` ثم تحسب `x%y`. الافضل حسابها مرة واحدة.

يأخذ الاقتران الجاهز `divmod` قرينتين و يرجع توبل بقيمتين، الناتج و الباقي، و يمكنك حفظ النتيجة كتوبل:

```
>>> t = divmod(7, 3)
>>> print t
(2, 1)
```

أو يمكنك استخدام تعيين لتوبل و تخزن العناصر بشكل مستقل:

```
>>> quot, rem = divmod(7, 3)
>>> print quot
```

```
2
>>> print rem
1
```

هذا مثال لاقتران يرجع توبل:

```
def min_max(t):
   return min(t), max(t)
```

max و min اقترانان جاهزان يجدان أكبر و أصغر عناصر تسلسل، أما min_max فيحسبهما و يرجع توبلا بالقيمتين معا.

12.4 قرينة طول-المتغير variable-length argument

بوسع الاقترانات أي عدد من القرائن، و البرمتر الذي يبدأ اسمه بـ * **يلملم** القرائن و يجعلها توبل، فمثلا printall يأخذ أي عدد من القرائن و يطبعها:

```
def printall(*args):
   print args
```

لبرمتر اللم أن يأخذ أي اسم، args كانت تقليدية. التالي يبين عمل الاقتران:

```
>>> printall(1, 20, '3')
(1, 20, '3')
```

المكمل للملمة هو التشتيت scatter. ان كان لديك تسلسلا من القيم و أردت تمريرها إلى اقتران كعدة قرائن، فيمكنك استخدام المؤثر *. فمثلا: لن تعمل divmod التي تأخذ قرينتين بالضبط في التوبل:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

أما إن شتَّت التوبل فإنها ستعمل:

```
>>> divmod(*t)
(2, 1)
```

تمرين 12.1 تستخدم العديد من الاقترانات الجاهزة توبلات قرائن طول-المتغير. فمثلا، max و min لهما أن يأخذا أي عدد من القرائن:

```
>>> max(1,2,3)
3
```

الا أن sum لا يفعل ذلك:

```
>>> sum(1,2,3)
TypeError: sum expected at most 2 arguments, got 3
```

اكتب اقترانا اسمه sumall يأخذ أي عدد من القرائن و يرجع مجموعها.

12.5 القوائم و التوبلات

يأخذ الاقتران الجاهز zip تسلسلين أو أكثر و يضغطها في قائمة أو توبل، بحيث يحتوي كل توبل على عنصر واحد من كل تسلسل. في بايثون3 يرجع zip تكرارا من التوبل ، لكن لمعظم الاهداف يتصرف التكرار كالقائمة.

هذا المثال يضغط محارف و قائمة:

```
>>> s = 'abc'
```

```
>>> t = [0, 1, 2]
>>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]
```

و نتيجته قائمة من التوبل كل منها يحتوي على حرف من المحارف و العنصر المقابل له من القائمة.

إن لم تكن التسلسلات بنفس الطول ستكون النتيجة بطول أقصرها:

```
>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

و يمكنك استخدام تعيين توبل في حلقة for للمرور في قائمة من التوبل:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
   print number, letter
```

في كل دورة في الحلقة، يختار بايثون التوبل التالي في القائمة و يعين العناصر لـ letter و number. ناتج هذه الحلقة هو:

```
0 a
1 b
2 c
```

إن جمعت بين for و zip و تعيين توبل ستحصل على تركيبة لغوية مفيدة للمرور في تسلسلين (أو أكثر) بنفس الوقت فمثلا has_match يأخذ تسلسلين t1 و t2 و يرجع True إن عثر على مؤشر بحيث تكون t1[i]=t2[i] :

```
def has_match(t1, t2):
   for x, y in zip(t1, t2):
      if x == y:
         return True
   return False
```

ان احتجت المرور على عناصر تسلسل و مؤشراتها، يمكنك استخدام الاقتران الجاهز enumerate :

```
for index, element in enumerate('abc'):
   print index, element
```

و مرة اخرى، مخرجات هذه الحلقة هي:

```
0 a
1 b
2 c
```

12.6 القواميس و التوبلات

للقواميس طريقة تدعى items ترجع قائمة من التوبل، يكون كل توبل زوجي مفتاح-قيمة.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

و كما تتوقع من القواميس فالعناصر أُرجعت بدون ترتيب معين. في بايثون3 items ترجع تكرارا، و لمعظم الغايات فالتكرار يتصرف كالقائمة.

و ان سرت في الاتجاه الاخر ستتمكن من استخدام قائمة توبل لاستهلال قاموس جديد:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
```

```
>>> d = dict(t)
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

و إن جمعت بين zip و dict ستكتشف طريقة موجزة لإنشاء القواميس:

```
>>> d = dict(zip('abc', range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

و من طرق القواميس، الطريقة update التي تأخذ قائمة من التوبل و تضيفها كزوجي مفتاح-قيمة إلى القاموس الموجود.

و الجمع بين items و تعيين توبل و for سيُكوِّن صيغة مرور بمفاتيح و قيم القاموس:

```
for key, val in d.items():
   print val, key
```

و المخرجات هي:

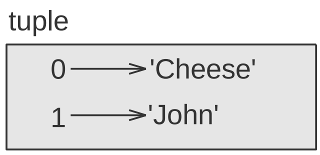
```
0 a
2 c
1 b
```

مرة أخرى.

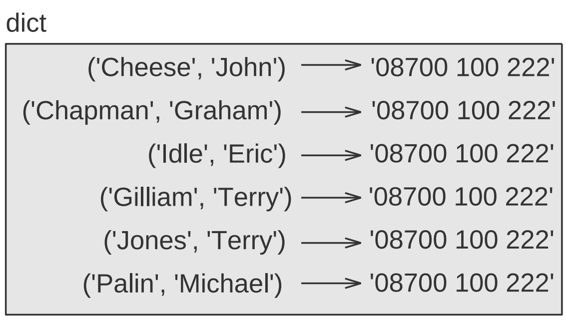
من الشائع استخدام التوبل كمفاتيح في القواميس (السبب الرئيسي هو عدم القدرة على استعمال القوائم). فمثلا قد يخطط دليل هاتف بين الاسم الاخير و الاسم الاول كزوجين و بين رقم الهاتف فعلى افتراض أننا عينا last, first و number فسيمكننا كتابة:

```
directory[last,first] = number
```

التعبير الموجود بين قوسين هو توبل. يمكننا استخدام تعيين توبل للمرور بهذا القاموس.



الشكل 12.1: رسم الحالة



الشكل 12.2: رسم الحالة

```
for last, first in directory:
   print first, last, directory[last,first]
```

هذه الحلقة تمر بالمفاتيح في dictionary و التي هي عبارة عن توبل، و تعين العناصر في كل توبل إلى first و last ، ثم الاسم و رقم الهاتف المقابل له.

يعبر عن التوبل بطريقتين في رسم الحالة النسخة المفصلة تظهر المؤشرات و العناصر تماما كما تظهر في القائمة مثلا، التوبل

`('Cheese' , 'John')` ستظهر كما في الشكل 12.1.

أما في الرسوم البيانية الضخمة فمن الأفضل التخلي عن التفاصيل. مثلا، الرسم لدليل الهاتف يبدو كما في الشكل 12.2.

و هنا تظهر التوبل باستخدام نحو بايثون كاختزال تصويري.

رقم الهاتف الظاهر في الرسم هو رقم الشكاوى في BBC لذا رجاءا لا تستخدمه.

12.7 مقارنة التوبل

مؤثرات النسبة تعمل على توبل و التسلسلات الاخرى. يبدأ بايثون بمقارنة أول عنصر من كل تسلسل، فإن كانت متساوية فسينتقل إلى العناصر التالية و هكذا، حتى يعثر على عناصر متخالفة. العناصر التي تلي ذلك التخالف تهمل (حتى و إن كانت كبيرة جدا):

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

اقتران `sort` يعمل بنفس الطريقة فإنه في الاساس يرتب العناصر حسب أولها، لكن في حالة العقد، سيرتبها حسب العنصر الثاني، و هكذا

تمنح هذه الميزة نفسها إلى نمط يقال له DSU و هو لـ:

تزيين Decorate تسلسل ببناء قائمة توبل بواحد أو أكثر من المفاتيح التي تسبق العناصر من التسلسل،

فرز Sort قائمة من التوبل، و

إزالة الزينة Undecorate بانتزاع العناصر المرتبة من التسلسل.

مثلا، افرض ان لديك قائمة كلمات و تريد ترتيبها من الاطول إلى الاقصر:

```
def sort_by_length(words):
   t = []
   for word in words:
      t.append((len(word), word))

   t.sort(reverse=True)

   res = []
   for length, word in t:
      res.append(word)
   return res
```

الحلقة الاولى تنشئ قائمة من التوبلات يكون كل توبل منها الكلمة و يسبقها طولها.

تقارن `sort` أول عنصر، `length`، أولا، و تأخذ العنصر الثاني فقط لكسر الروابط. القرينة، الكلمة المفتاحية `reverse = True`، تأمر `sort` أن يستمر بترتيب تنازلي.

الحلقة الثانية تمر في قائمة التوبل و تنشئ قائمة من الكلمات مرتبة تنازليا.

تمرين 12.2 في المثال السابق كُسرت الروابط عن طريق مقارنة الكلمات، فالكلمات المتساوية الطول ظهرت مرتبة ترتيبا أبجديا معكوسا. قد تتطلب تطبيقات أخرى كسر الروابط عشوائيا. عدل هذا المثال بحيث تظهر الكلمات متساوية الطول مرتبة

عشوائيا. تلميح: أنظر الاقتران `random` في مديول `random`. الحل:
.http://thinkpython.com/code/unstable_sort.py

12.8 تسلسلات التسلسلات

لقد ركزت على القوائم و التوبل، لكن تقريبا كل الامثلة في هذا الفصل ستعمل مع قوائم القوائم، توبل التوبل و توبل القوائم، من الافضل أحيانا الحديث عن تسلسلات التسلسلات لتجنب سرد كل التركيبات الممكنة.

يمكن استخدام أنواع التسلسلات المختلفة (المحارف، القوائم و التوبل) بالتبادل في معظم السياقات، فلماذا و كيف تختار أحدها دون الاخر؟

و لكي نبدأ بالاوضح، المحارف، لأنها محددة أكثر من باقي التسلسلات، حيث يجب أن تكون العناصر حروفا، و هي تقريبا ثبيتة. فإن كان هدفك استبدال الحروف في المحارف فالافضل استخدام قائمة من الحروف.

القوائم شائعة أكثر من التوبل، لأنها تتبدل على الاغلب. إلا أن هناك بضعة حالات قد تجعلك تفضل التوبل:

1. في بعض السياقات يكون من الابسط دلاليا انشاء التوبل كما في عبارة `return` و في حالات أخرى قد تفضل القوائم.
2. إن اردت استخدام تسلسلا كمفاتيح في قاموس، فعليك استخدام نمط ثبيت كالمحارف أو التوبل.
3. و إن كنت تمرر تسلسلا كقرينة إلى اقتران، سيكون من الآمَن استخدام التوبل فاحتمال ظهور سلوك غير متوقع بسبب تعدد المرجعيات أقل.

و لأن التوبلات ثبيتة فليس لها طرق مثل `sort` و `reverse` التي تعدل القوائم الموجودة. الا أن بايثون يوفر اقترانات جاهزة `sorted` و `reversed`، و التي تأخذ أي تسلسل كبرمتر و ترجع قائمة جديدة بها نفس العناصر بترتيب اخر.

12.9 علاج الأخطاء

القوائم و القواميس و التوبل تعرف عامة بـ data structure. في هذا الفصل بدأنا برؤية هياكل بيانات مركبة كقوائم التوبل، و قواميس مفاتيحها توبل و قيمها قوائم. هياكل البيانات المركبة مفيدة الا أنها مرتع لما أسميه الأخطاء الشكلية، أعني عندما يكون لهياكل البيانات النوع أو الحجم أو التركيب الخطأ، فإن كنت تتوقع منى قائمة بها عدد صحيح واحد و أعطيك عدد صحيح بحت قديم (ليس في قائمة) فلن تعمل.

و للمساعدة في علاج أخطاء كهذه، كتبت مديولا اسمه `structshape`، يأخذ أي نوع من هياكل البيانات كقرينة و يرجع محارف تلخص شكل هيكل البيانات يمكنك تحميله من:
.http://thinkpython.com/code/structshape.py

ها هي نتيجة قائمة بسيطة:

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print structshape(t)
list of 3 int
```

لو كان البرنامج فاخرا لكتب `list of 3 int`**s**، لكن كان أسهل لي ألّا اتعامل مع صيغ الجمع.

و هذه قائمة قوائم:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> print structshape(t2)
list of 3 list of 2 int
```

ان لم تكن عناصر القائمة من نفس النوع، سيجمعها `structshape` بالترتيب حسب النمط:

```
>>> t3 = [1, 2, 3, 40, '5', '6', [7], [8], 9]
>>> print structshape(t3)
list of (3 int, float, 2 str, 2 list of int, int)
```

و هذه قائمة من التوبل:

```
>>> s = 'abc'
>>> lt = zip(t, s)
>>> print structshape(lt)
list of 3 tuple of (int, str)
```

و هذه قاموس به ثلاثة عناصر تخطط الاعداد الصحيحة إلى محارف:

```
>>> d = dict(lt)
>>> print structshape(d)
dict of 3 int->str
```

فإن كنت تواجه صعوبات في ملاحقة هياكل بياناتك سيساعدك `structshape`.

12.10 المعاني

توبل Tuple: تسلسل ثبيت من العناصر.

تعيين لتوبل Tuple assignemt : تعيين يكون التسلسل فيه على اليمين و متغيرات توبل على اليسار يقيم الجانب الايمن ثم تعين عناصره للمتغيرات على الجانب الايسر.

الملمة gather : عملية تجميع توبل قرينة طول- المتغير.

تشتيت scatter : معاملة تسلسل كقائمة من القرائن.

DSU : إختصار Decorate-Sort-Undecorate أي زين-رتب-أزل الزينة. قالب عملياتي يتم فيه انشاء قائمة توبل، ترتيبها و انتزاع قسم من النتيجة.

هيكل بيانات Data structure : تجمع لقيم ذات صلة، تكون في الغالب مرتبة في قوائم أو توبل أو قواميس ... الخ.

شكل (هيكل بيانات) shape (of a data structure) : تلخيص لنمط و حجم و تركيبة هيكل البيانات.

12.11 تمارين

تمرين 12.3 أكتب اقترانا اسمه `most_frequent` يأخذ محارف و يطبع حروفها تنازليا حسب حجم تكرارها. احصل على عينات لنصوص من عدة لغات و راقب تغير ترددات الحروف في كل لغة، قارن نتيجتك بالجدول: `http://enwikipedia.org/wiki/Letter_frequencies`.

الحل: `http://thinkpython.com/code/most_frequent.py`.

تمرين 12.4 مزيدا من الجناس اللفظي anagram:

1. أكتب برنامجا يقرأ قائمة كلمات من ملف (انظر القسم 91) ثم يطبع كمجموعات الكلمات التي بها جناس لفظي.
هنا مثال لما يجب أن يظهر عليه المخرج:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

2. عدل المثال السابق بحيث يطبع أكبر مجموعات الجناس اللفظي أولا، ثم يتبعها ثاني أطول المجموعات و هكذا.
3. في لعبة الحروف المبعثرة. تكون bingo عندما تلعب كل الحروف السبعة التي على الرف مع الحرف الذي على اللوحة، لتكون كلمة ثمانيّة الحروف. أيُّ مجموعة من ثمانية حروف ستمكنك من عمل أكبر عدد Bingo ممكن؟

تلميح: there are seven. الحل: `http://thinkpython.com/code/anagram_sets.py`.

تمرين 12.5 تسمى الكلمتين اللتين إن غيرت مكان حرفين من أيهما تغير الكلمة إلى الثانية metathesis pair مثل converse و conserve اكتب برنامجا يجد كل الازواج المطابقة لهذا المعنى في القاموس. تلميح: لا تختبر كل زوج من الكلمات و لا تختبر كل الاحتمالات لتبديل مكان حرفين في كل كلمة الحل:

`http://thinkpython.com/code/metathesis.py`.

عرفان: هذا التمرين مستلهم من مثال في `http://puzzlers.org`.

تمرين 12.6 مرة أخرى من أحجيات Car Talk:

`http://www.cartalk.com/content/puzzlers`.

ما هي أطول كلمات اللغة الانجليزية، و التي يجب أن تصلح ككلمة انجليزية كلما أزلت واحدا من حروفها في كل مرة؟

حسنا، بإمكانك حذف الحروف من أي ناحية تريد أو من وسط الكلمة، لكن لا يمكنك إعادة ترتيب الحروف. و في كل مرة تحذف حرفا ستواجه بكلمة انجليزية أخرى، و في الآخر ستنتهي إلى حرف واحد، و على هذا الحرف أن يكون كلمة انجليزية مقبولة -كلمة موجودة في المعجم. أريد أن أعرف ما هي أطول كلمة إنجليزية بهذا الشكل و ما عدد حروفها؟

سأعطيك مثالا متواضعا: Sprite ؟ ستبدأ بـ sprite ثم تحذف حرفا، لعله من الداخل، فلنحذف r فستتبقى لنا spite ثم نأخذ منها e ليتبقى spit ثم نحذف s فيتبقى pit ، ثم it و في النهاية I.

اكتب برنامجا يعثر على كل الكلمات التي ينطبق عليها هذا الشرط، ثم جد أطولها.

التحدي في هذا التمرين أكبر من سابقيه، لذا إليك بعض النصائح:

1. قد تود كتابة اقتران يأخذ كلمة و يحتسب كل الكلمات التي تولد منها بإزالة حرف واحد (تسمى الكلمات المنتجة children أبناء الكلمة).
2. إجترارىا، يمكن اختزال الكلمة إن أمكن إختزال أطفالها و كحالة قاعدة بإمكانك اعتبار أنه يمكن اختزال المحارف الفارغة.
3. لا تحتوي قائمة الكلمات التي حملتها `wordstxt` على كلمات من حرف واحد قد يكون عليك إضافة I و a و المحارف الفارغة.
4. لتحسين أداء برنامجك قد تكون بحاجة لاستخدام الـ memo لتجعل البرنامج يذكرك بالكلمات القابلة للإختزال.

الحل: `http://thinkpython.com/code/reducible.py`.

الفصل الثالث عشر

دراسة حالة: إختيار هياكل البيانات

13.1 تحليل تكرار الكلمات

كالعادة، عليك على الاقل محاولة حل التمارين التالية قبل قراءة حلي لها.

تمرين 13.1 اكتب برنامجا يقرأ ملفا، ثم يقسّم سطوره إلى كلمات، و يعري المسافات البيضاء و التشكيل، ثم يحول جميع الحروف إلى حروف صغيرة.

تلميح: مديول `string` يوفر محارف اسمها `whitespaces` و التي تحتوي على `space, tab, newline` الخ و كذلك `punctuation` التي تحتوي على التشكيل. لنر إن كان بوسعنا جعل بايثون يشتم:

```
>>> import string
>>> print string.punctuation
!"#$%&'()*+,-/:;<=>?@[\]^_`{|}~
```

و بإمكانك أخذ طرق المحارف `strip, replace` و `translate` بالاعتبار.

تمرين 13.2 قم بزيارة مشروع جوتنبرج `http://Gutenberg.org`، و حمل من هناك ما يروق لك من الكتب التي حقوقها أصبحت عامة. الكتب بصيغة ملفات نصية.

عدل برنامجك من التمرين السابق ليقرأ الكتاب الذي حملته، تخطى المعلومات التي في ترويسة الكتاب و عالج باقي الكلمات كما في السابق.

بعد ذلك عدل البرنامج ليعُدّ إجمالي كلمات الكتاب، و عدد المرات التي استعملت فيها كل كلمة.

إطبع عدد الكلمات المختلفة في الكتاب. قارن بين كتب مختلفة لكتّاب مختلفين في أزمنة مختلفة. أي من المؤلفين له أكبر إلمام بالالفاظ (كتابه يشمل عدد أكبر من الالفاظ المختلفة).

تمرين 13.3 عدل على البرنامج السبق ليطبع أكثر 20 كلمة مكررة في الكتاب.

تمرين 13.4 عدل البرنامج السابق ليقرأ قائمة كلمات (أنظر القسم 9.1) ثم يطبع كل كلمات الكتاب الغير موجودة في قائمة الكلمات. كم عدد الأخطاءالمطبعية؟ كم عدد الكلمات الشائعة التي يجب أن تورد في قائمة الكلمات. و كم عدد الكلمات الغامضة؟

13.2 الأعداد العشوائية

إن أُعطِيَت الحواسيب نفس المدخلات فستنتج نفس المخرجات في كل مرة، لهذا يقال لها حتمية. الحتمية في الغالب صفة محمودة، طالما أننا نتوقع أن تنتج لنا نفس الحسبة النتيجة نفسها، لكن في بعض التطبيقات نريد للحاسوب أن يفاجئنا، الالعاب مثال واضح لهذه التطبيقات، لكن هناك أكثر.

لقد تبين أن عمل برنامج غير حتمي حقيقي ليس بالامر السهل، إلا أن هناك سبل لجعله يبدو غير حتمي على الاقل، منها استخدام خوارزميات تولد تقليدا للأعداد العشوائية، رغم أنها ليست أعدادا عشوائية، لكن بمجرد نظر إليها سترى أنه من المستحيل التفريق بينها و بين العشوائية.

مديول `random` يولد هذه الاعداد العشوائية المقلدة (من الان سأدعوها الارقام العشوائية).

مديول `random` يرجع أرقاما عامة عشوائية بين 0.0 و 1.0 (من ضمنها 0.0 و لكن ليس 1.0) و في كل مرة تنادي فيها `random` ستحصل على رقم في سلسلة طويلة و لتر عينة، شغل هذا البرنامج:

```
import random

for i in range(10):
   x = random.random()
   print x
```

الاقتران `randint` يأخذ البرمترين `low` و `high` ثم يرجع عددا صحيحا بين `low` و `high` (و قد يكون أحدهما).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

و لتختار عنصرا من قائمة عشوائيا بإمكانك استخدام `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

و يوفر مديول `random` اقترانات لتوليد قيم عشوائية من توزيعات مستمرة كـ Gaussian, exponential, gamma و بضعة توزيعات أخرى.

تمرين 13.5 اكتب اقترانا اسمه `choose_from_hist` يأخذ مدرج تكراري كالذي شرح في القسم 11.1 و يرجع قيما عشوائية من المدرج، و تكون مختارة باحتمال يتناسب مع ترددها مثلا، لهذا المدرج التكراري:

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> print hist
{'a': 2, 'b': 1}
```

سيرجع اقترانك `a` باحتمال 2/3 و `b` باحتمال 1/3.

13.3 مدرج تكراري للكلمات

عليك المحاولة مع التمارين السابقة قبل التقدم، بإمكانك تحميل حلولي من:

.http://thinkpython.com/code/analyze_book.py

و ستحتاج أيضا إلى http://thinkpython.com/code/emma.txt.

إليك برنامجا يقرأ ملفا و ينشئ مدرجا تكراريا من كلماته:

```
import string


def process_file(filename):
   hist = dict()
   fp = open(filename)
   for line in fp:
      process_line(line, hist)
   return hist

def process_line(line, hist):
   line = linereplace('-', ' ')

   for word in line.split():
      word=word.strip(string.punctuation + string.whitespace)
      word = wordlower()

      hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

هذا البرنامج يقرأ emma.txt الذي يحتوي نص "إمة" لجين إوستن

يدور process_file في سطور الملف، و يمررها واحد تلو الاخر إلى process_line. و يُستخدم المدرج التكراري hist كمراكم.

يستخدم process_line طريقة المحارف replace لاستبدال الوصلات (-) بفراغات قبل استخدام split لتقسيم السطر إلى قائمة من المحارف. ثم تمر في قائمة الكلمات و تستخدم strip و lower لإزالة التشكيل و لتحويل كل الحروف إلى حروف صغيرة (كان تعبير "تحويل كل الحروف" للإختصار، تذكر بأن المحارف ثبيتة، فطرق كـ strip و lower ترجع محارف جديدة).

و أخيرا، يحدِّث process_line المدرج التكراري، إما عن طريق إنشاء عنصر جديد أو الزيادة على عنصر موجود.

و لعد كلمات الملف يمكننا تجميع الترددات في المدرج التكراري:

```
def total_words(hist):
   return sum(hist.values())
```

و عدد الكلمات المختلفة هو مجرد عدد كلمات الملف:

```
def different_words(hist):
   return len(hist)
```

و هنا نص برمجي يطبع النتائج:

```
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

و النتائج:

```
Total number of words: 161080
Number of different words: 7214
```

13.4 أكثر الكلمات ورودا

لنجد أكثر الكلمات استخداما، يمكننا تطبيق ال DSU، تأخذ most_common مدرجا تكراريا و ترجع قائمة توبل من الكلمة-ترددها، و تكون مرتبة عكسيا حسب التردد:

```
def most_common(hist):
   t = []
   for key, value in hist.items():
      t.append((value, key))
   t.sort(reverse=True)
   return t
```

و هذه حلقة تطبع أكثر عشر كلمات ورودا:

```
t = most_common(hist)
print 'The most common words are:'
for freq, word in t[0:10]:
   print word, '\t', freq
```

و هذه النتائج من نص "إمة" :

```
The most common words are:
to 5242
the 5205
and 4897
of 4295
i 3191
a 3130
it 2529
her 2483
was 2400
she 2364
```

13.5 البرمترات الاختيارية

لقد رأينا اقترانات تأخذ أعدادا مختلفة من القرائن. من الممكن كتابة اقترانات يعرفها المستخدم (user-defined) و تكون قرائنها اختيارية كذلك. فمثلا، هذا الاقتران يطبع أكثر الكلمات استخداما في مدرج تكراري:

```
def print_most_common(hist, num=10):
      t = most_common(hist)
      print 'The most common words are:'
      for freq, word in t[:num]:
            print word, '\t', freq
```

البرمتر الأول أساسي، بينما الثاني إختياري و القيمة الافتراضية لـ num هي 10.

و إن زودته بقرينة واحدة:

```
print_most_common(hist)
```

فستأخذ num القيمة الافتراضية، و إن زودته بالقيمتين:

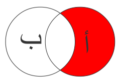
```
print_most_common(hist, 20)
```

فستأخذ num قيمة القرينة. بكلمات أخرى فالقرينة الاختيارية تقدم على الافتراضية.

إن كان الاقتران يأخذ برمترات اختيارية و اخرى مطلوبة، فسيقدم المطلوبة على الاختيارية ثم تتبعها الاختيارية.

13.6 عمليات الطرح في القواميس

مسألة العثور في القاموس على كلمات لا توجد في word.txt قد تذكرك بالمجموعات المكملة، أي أننا نريد معرفة كل الكلمات الموجودة في مجموعة ما (و هي التي في الكتاب) و غير الموجودة في مجموعة أخرى (و هي الكلمات التي في القائمة).



{1, 2, 3} \ {2, 3, 4} = {1}

تأخذ subtract القاموسين d1 و d2 و ترجع قاموسا جديدا يحتوي على كل المفاتيح الموجودة في d1 و لكن ليست موجودة في d2، و طالما أننا لا نكترث لقيمها فسنجعلها كلها None:

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

و لنجد الكلمات التي في الكتاب و لا توجد في words.txt، بوسعنا استخدام process_file لبناء مدرج تكراري لـ words.txt ثم نطرح:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print "The words in the book that aren't in the word list are:"
for word in diff.keys():
    print word,
```

إليك بعض النتائج من "إمة":

```
The words in the book that aren't in the word list are:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

تمرين 13.6 يوفر لنا بايثون هيكل بيانات يسمى set و به العديد من عمليات المجموعات المعروفة. اقرأ وثائقه على http://docspython.org/2/library/stdtypes.html#types-set.

ثم اكتب برنامجا يستخدم طرح المجموعات للعثور على كلمات الكتاب الغير موجودة في قائمة الكلمات الحل: http://thinkpython.com/code/analyze_book2.py.

13.7 الكلمات العشوائية

ابسط خوارزمية لاختيار كلمة من مدرج تكراري عشوائيا هي بناء قائمة بها نسخ متعددة لنفس الكلمة، عدد النسخ يتناسب مع تردد ورودها حسب المدرج التكراري، و بعدها نختار من هذه القائمة:

```
def random_word(h):
   t = []
   for word, freq in h.items():
      text.end([word] * freq)

   return random.choice(t)
```

التعبير `freq * [word]` يوجِد قائمة من نسخ المحارف `word` عددها `freq`. أما الطريقة `extend` فهي شبيهة بـ `append` ما عدا أن القرائن هي تسلسلات.

تمرين 13.7 قامت الخوارزمية السابقة بعملها، لكن من الواضح عدم فعاليتها: فكل مرة تختار فيها كلمة عشوائية ستعيد بناء القائمة، و التي هي بحجم الكتاب. من الواضح أن تحسينها يكمن في بناء القائمة مرة واحدة ثم القيام باختيارات متعددة، لكن القائمة ستبقى كبيرة.

البدائل:

1. استخدم `keys` فسيكون لديك قائمة بكلمات الكتاب.
2. انشئ قائمة تحتوي مجموع تردد الكلمات (انظر التمرين 10.3). آخر عنصر في هذه القائمة هو مجموع أعداد الكلمات في الكتاب، `n`.
3. اختر رقما عشوائيا بين `1` و `n` و استخدم تنصيف البحث (أنظر تمرين 11.10) لتجد المؤشر حيث سيغرز الرقم العشوائي في المجموع التراكمي.
4. استخدم المؤشر لتجد الكلمة المقابلة في القائمة.

اكتب برنامجا يستعمل هذه الخوارزمية لاختيار كلمة عشوائية من الكتاب الحل:

.http://thinkpython.com/code/analyze_book3.py

13.8 تحليل ماركوف

ان اخترت الكلمات عشوائيا من كتاب، فجُلّ ما ستحصل عليه هو بعض الالفاظ، أما جملة صحيحة فلا:

`this the small regard harriet which knightley's it most things`

"هذا القليل نسبة هارييت التي للفروسية فتلك أكثر أشياء"

استرسال الكلمات العشوائية نادرا ما يعقل، فلا توجد علاقة منطقية بين الكلمات المتعاقبة، فمثلا ستتوقع بعد الفعل فاعل أو اسم فاعل و لن تتوقع "هل".

لقياس علاقات كهذه هناك تحليل ماركوف الذي يصف العلاقة بين الكلمة و التي تليها مثلا، أغنية *Eric, the half a bee:*

Half a bee, philosophically,
Must, ipso facto, half not be
But half the bee has got to be
Vis a vis, its entity D'you see?
But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee

Due to some ancient injury?

و تعني (بالعربية):

نصف نحلة، فلسفيا،
هو، في الحقيقة، نصف لا يكون.
لكن على نصف النحلة أن يكون
وجها لوجه،كيانها. أفترى؟
لكن هل لنحلة قيل أنها كانت
أو لا تكون بالكامل نحلة
لمّا كان نصف نحلة ليس نحلة
ربما لجرح قديم؟

في هذا النص أُتبعت half the بالكلمة bee دائما، لكن العبارة the bee أتبعت أحيانا بـ has و أخرى بـ is.

كانت نتيجة تحليل ماركوف، هي الوصل بين البادئات كـ half the و the bee و بين اللاحقات كـ has و is.

إن أُعطيت هذه التخطيط، فسيمكنك إنتاج نص عشوائي إن ابتدأت بأي بادئة ثم اخترت عشوائيا أي لاحقة محتملة، و بعدها يمكنك تركيب نهاية البادئة مع اللاحقة الجديدة لخلق بادئة جديدة، ثم تعيد الكَرَّة.

مثلا: ان ابتدأت ببادئة Half a فالكلمة التالية يجب أن تكون bee، لأن البادئة تظهر مرة واحدة في النص، و البادئة التالية ستكون a bee و عليه فستكون اللاحقة التالية إما phylosophically أو be أو due.

كان طول البادئة في هذا المثال 2 دائما، الا أنه يمكنك عمل تحليل ماركوف بأي طول لبادئة يسمى طول البادئة ""ترتيب التحليل."

تمرين 13.8 تحليل ماركوف

1.اكتب برنامجا يقرأ النص من ملف و يجرى تحليل ماركوف عليه على النتيجة أن تكون قاموسا يصل بين البوادئ و بين اللواحق المحتملة للمجموعة أن تكون قائمة أو توبل أو قاموس، الامر راجع لك لتقوم بأنسب الاختيارات. يمكنك اختبار برنامجك بالطول 2 للبادئة، لكن عليك كتابة البرنامج بطريقة تجعل اختبار أطوال اخرى ممكنا.

2.اضف اقترانا للبرنامج السابق لتولد نصا عشوائيا بناءا على تحليل ماركوف. خذ مثالا من "إمة" بطول البادئة :2

He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "Icannot make speeches, Emma:" he soon cut it all himself.

في هذا المثال تركت إشارات الوصل موجودة على الكلمات، النحو في النتيجة كان صحيحا، لكن ليس جدا. دلالية النص المنتج مقبولة، لكن ليس جدا. ما الذي سيحدث إن زدت طول البادئة؟ هل سيصبح النص

العشوائي معقولا أكثر ؟

3.بعد أن يصبح برنامجك شغالا، قد تود تجربة الخلطات: إن حلَّلت نصوصا من كتابين إضافيين فستتولد نصوص بتوليفة من الالفاظ و الصيغ من هذه المصادر بشكل مسل.

عرفان: دراسة الحالة هذه مبنية على مثال من "الممارسة البرمجية" لكِرِنجان و بايك.

عليك المحاولة في هذا التمرين قبل المضي قدما، و بعدها يمكنك تحميل حلي له:

.http://thinkpython.com/code/markov.py

سوف تحتاج أيضا إلى http://thinkpython.com/code/emma.txt.

13.9 هيكلة بيانات

من المسلي استخدام تحليل ماركوف لإنشاء نصوص عشوائية، لكن هناك هدف لهذا التمرين: هيكلة البيانات. في سياق حلِّك للتمارين السابقة كان عليك اختيار:

- كيف تمثل البوادئ.
- كيف تمثل مجموعات اللواحق المحتملة.
- كيف تمثل التخطيط من كل بادئة إلى مجموعة اللواحق المحتملة.

الخيار الاخير سهل، فنمط التخطيط الوحيد الذي رأيناه هو القاموس، فكان الخيار الطبيعي.

أما بالنسبة للبوادئ فالخيار المنطقي كان المحارف أو توبل المحارف. لكن للواحق هناك خياران، أحدهما القائمة و الثاني التردد المعياري (قاموس).

كيف تختار؟ عليك أولا التفكير بالعمليات التي ستحتاج إلى تطبيقها على كل هيكل بيانات. فللبوادئ تلزمنا القابلية لحذف كلمة من البداية و إضافتها للنهاية مثلا، إن كانت البادئة الحالية هي Half a و الكلمة التالية هي bee يلزم أن تكون قادرا على حذف البادئة التالية a bee.

قد يكون الخيار الاول قائمة، لسهولة إضافة و حذف العناصر، لكننا نحتاج أيضا إلى القدرة على استخدام البوادئ كمفاتيح في قاموس، إذن لم تعد القوائم خيارا مناسبا. مع التوبل لا يمكنك الاضافة أو الحذف، لكن باستخدام رمز الجمع تستطيع إنشاء توبل جديد:

```
def shift(prefix, word):
   return prefix[1:] + (word,)
```

تأخذ shift توبل من الكلمات prefix و محارف word و تكون توبل جديد به كل الكلمات في prefix ما عدا الاولى، و مضاف إلى نهايته word.

العلميات المطلوبة على مجموعات اللواحق تتضمن إضافة لاحقة جديدة (أو زيادة تردد لاحقة موجودة)، ثم اختيار لاحقة عشوائية.

إضافة لاحقة جديدة سهل في تطبيق التردد المعياري و سهل في تطبيق القوائم، اختيار عنصر عشوائي في قائمة سهل، الاختيار بكفاءة من تردد معياري أصعب (أنظر تمرين 13.7).

حتى اللحظة نحن نتكلم عن السهولة في التطبيق ، لكن هناك عوامل أخرى يجب أن تؤخذ في الاعتبار لاختيار هيكل

بيانات، منها زمن التشغيل. أحيانا يكون هناك سبب نظري للتوقع بأن هيكل بيانات ما أسرع من غيره. مثلا، قلت سابقا أن المؤثر `in` أسرع في القواميس منه في القوائم، على الاقل عندما يكون عدد العناصر كبيرا.

الا أنك في الغالب لن تعرف مسبقا أي التطبيقات سيكون اسرع. أحد الحلول هو تطبيق كلا الخيارين و رؤية أيهما أفضل، يسمى هذا المنحى بـ benchmark أو مقياس الأداء، أما البديل العملي فهو اختيار هيكل البيانات الذي يكون تحقيقه أسهل، ثم ترى إن كان سريعا كفاية بالنسبة للتطبيق المقصود، إن كان كذلك فلا داعي للإستمرار، و إلا فهناك أدوات مثل مديول `profile` الذي يظهر المواقع المستهلِكة للوقت في البرنامج.

العامل الاخر الذي يجب أخذه بالاعتبار هو حجم التخزين، فاستخدام مدرج تكراري لمجموعات اللواحق مثلا سيحتل مساحة أصغر لأنك ستخزن كل كلمة مرة واحدة فقط، بغض النظر عن عدد مرات ظهورها في النص. في بعض الحالات يجعل توفير المساحة برنامجك أسرع، و في أقصى حالات استنفاذ الذاكرة قد لا يعمل برنامجك بالمرة. لكن في معظم التطبيقات يكون حجم التخزين اعتبارا ثانويا بعد سرعة التشغيل.

و خاطرة أخيرة: طوال هذه المناقشة كنت أتحدث عن استخدام هيكل بيانات واحد للتحليل و الإنشاء، لكن طالما أن هاتان مرحلتان منفصلتان، فيمكننا أيضا استخدام هيكل بيانات للتحليل و نحوِّل الآخر للإنشاء. سيكون هذا ربحا صافيا إن كان الوقت الموفر خلال الانشاء أكبر من الوقت المستنفذ في التحويل.

13.10 علاج الأخطاء

هنالك أربعة أشياء يمكنك تجربتها عندما تحاول علاج بقة شرسة:

القراءة: تفحص نصك البرمجي و تأكد أنه يقول ما قصدت قوله.

التشغيل: إختبره بأن تقوم بتغييرات، و بأن تشغل عدة نسخ. غالبا ما تتّضح المشكلة عندما تعرض الشيء الصحيح في المكان الصحيح. لكن في أحيان أخرى قد يتطلب الامر منك استهلاك بعض الوقت في بناء السقالات.

إعادة التفكير: خذ بعض الوقت لتفكر! أي نوع من الأخطاء هو، نحوي، دلالي أم خطأ عند التشغيل؟ ما هي المعلومات التي تستقيها من رسالة وجود الخطأ، أو من مخرجات البرنامج؟ أي نوع من الأخطاء ينتج المشكلة التي بين يديك؟ ما هو اخر شيء عدلته قبل ظهور المشكلة؟

التراجع: في بعض المواقف تكون الحكمة هي التراجع، إعادة التعديلات الاخيرة إلى وضعها السابق واحدة تلو الاخرى إلى أن يختفي الخطأ و تعود إلى البرنامج الذي كنت تفهمه. ثم ابدأ البناء من تلك النقطة.

يَعْلق مبتدئي البرمجة أحيانا في واحدة من هذه النشاطات و ينسون النشاطات الاخرى. كل نشاط منها يأتي و معه نمط فشله.

مثلا قراءة برنامجك ستساعد إن كان الخطأ مطبعيا، لكن ليس إذا كان الخطأ عدم فهم المبدأ. فإن لم تفهم ما يقوم به برنامجك فلن تر الخطأ و لو قرأته 100 مرة، لأن الخطأ في رأسك.

إجراء التجارب قد يساعد، خصوصا إن جربت فحوصا صغيرة و بسيطة. لكن اجراءها بدون قراءة النص أو التفكير فيه سيوقعك فيما أسميه "برمجة المشي على غير هدى" و هي عملية القيام بتغييرات عشوائية إلى أن يقوم البرنامج بالعمل الصحيح. لا داعي للتذكير بأن بأن البرمجة على غير هدى ستتطلب الكثير من الوقت.

عليك إنفاق الوقت في التفكير. علاج الأخطاء كالعلوم التجريبية. فيجب أن يكون لديك فرضية واحدة على الاقل عن ماهية المشكلة، إن كانت لديك أكثر من فرضية ففكر باختبار يحذف إحداها.

أخْذُ استراحة يساعد في التفكير. و كذلك الحديث، إن شرحت المشكلة إلى أحدهم (قد تكون أنت) فأحيانا ستجد الحل حتى قبل إنهاء الشرح.

إن كثرت الأخطاء في البرنامج فلن تساعدك حتى أعتى طرق علاج الاخطاء، و كذلك إن كان النص كبير جدا و معقدا جدا. أحيانا الخطو إلى الخلف يكون حلا، بسَّط البرنامج إلى الحد الذي يعمل فيه بشكل صحيح و تفهمه.

يتردد مبتدئي البرمجة في التراجع، لعدم قبولهم حذف سطرا من نص برمجي كانوا قد كتبوه. إذا كان نسخ البرنامج إلى ملف جديد يواسيك، فقم بذلك و ابدأ بتعريته. يمكنك بعدها لصق قطع البرنامج واحدة تلو الاخرى.

يتطلب العثور على بقة عتية منك القراءة و التشغيل و إعادة التفكير، و أحيانا التراجع. فإن علِقت في إحدى هذه النشاطات اتركها و جرب الاخريات.

13.11 المعاني

حتمي deterministic: البرنامج الذي يقوم بنفس الشيء كلما شُغِّل، إن أعطي نفس المدخلات.

الارقام العشوائية المزيفة pseudorandom: ذلك التسلسل من الارقام الذي يبدو عشوائيا الا أنه وجد عن طريق برنامج حتمي.

القيمة الافتراضية default value: هي القيمة التي تعطى لبرمتر اختياري إن لم يزوَّد بقرينة.

قياس الأداء Benchmarking: عملية الاختيار بين هياكل البيانات عن طريق تحقيق بدائل ثم فحصها على عينة من المدخلات المحتملة.

13.12 تمارين

تمرين 13.9 منزلة الكلمة rank هو موقعها في قائمة من الكلمات المرتبة حسب التردد: فأكثر الكلمات ترددا لها المنزلة 1 و ثاني أكثر الكلمات ترددا لها المنصب 2... الخ.

قانون زِف يصف العلاقة بين منزلة و تردد الكلمات في اللغات الطبيعية

.http://en.wikipedia.org/wiki/Zipf's_law

تحديدا فهو يتوقع أن يكون التردد f للكلمة التي منصبها r هو:

$$f = cr^{-s}$$

حيث s و c هي برمترات تعتمد على اللغة و على النص إن أخذت خوارزمية كلا جانبي المعادلة ستحصل على:

$$\log f = \log c - s \log r$$

فإن رسمت منحنى $\log f$ و $\log r$ ستحصل على خط مستقيم له ميل $-s$ و يتقاطع مع $\log c$.

اكتب برنامجا يقرأ النص من ملف، ثم يعد الترددات و يطبع سطرا لكل كلمة بترتيب تنازلي يكون فيه $\log f$ و $\log r$. استخدم برنامج الرسوم البيانية الذي تفضل لرسم النتيجة، ثم انظر ان كانت تشكل خطا مستقيما. هل يمكنك توقع قيمة s؟

الحل: .http://thinkpython.com/code/zipf.Py

و لرسم المنحنى قد تحتاج إلى تنصيب .http://matplotlib.sourceforge.net

الفصل الرابع عشر

الملفات

14.1 الثبات persistance

معظم البرامج التي رأيناها حتى الان مؤقتة، بمعنى أنها تعمل لوقت قصير و تنتج بعض البيانات، ثم عند توقفها تختفي بياناتها و إن شغلت البرنامج من جديد فسيبدأ بداية نظيفة.

البرامج الاخرى ثابتة: تعمل لوقت طويل (أو طوال الوقت)، تحتفظ ببعض بياناتها على الاقل في مخزِّنات دائمة (كالقرص الصلب) و إن أوقفت و أعيد تشغيلها ستتابع من حيث توقفت.

أنظمة التشغيل مثال على البرامج **الثابتة** (المثابرة) ، فهي تبدأ مع تشغيل الحاسوب أو خادم الويب (web server) التي تعمل طوال الوقت منتظرة دائما الاوامر من الشبكة.

من ابسط طرق احتفاظ البرامج ببياناتها هي قراءة و كتابة ملفات النصوص، لقد استخدمنا من قبل برامج تقرأ و تكتب ملفات النصوص. سنرى في هذا الفصل برامج تكتب هذه البرامج.

الطرق البديلة هي تخزين حالة البرنامج في قاعدة بيانات. في هذا الفصل سأقدم قاعدة بيانات بسيطة و مديول `pickle` الذي يسهل تخزين بيانات البرنامج.

14.2 القراءة و الكتابة

الملف النصي هو تسلسل من الحروف المخزنة بشكل دائم على وسيط كالقرص الصلب، أو ذاكرة فلاش أو قرص مدمج. مرَّ بنا كيفية فتح و قراءة ملف في القسم 9.1.

لكتابة ملف عليك أولا فتحه في الوضع `w` كالبرمتر الثاني:

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

إن كان الملف موجودا من قبل، ففتحه بوضع الكتابة سيمحو البيانات القديمة و يبدأ بداية نظيفة، لذلك كن حذرا! و إن لم يكن الملف موجودا من قبل فسيوجد ملف جديد.

طريقة `write` تضع البيانات في الملف:

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

و أكرر بأن كائن الملف يحتفظ بآخر موقع كان فيه، فإن نوديت `write` مرة أخرى ستضيف البيانات إلى نهاية الملف:

```
>>> line2 = "the emblem of our land\n"
>>> fout.write(line2)
```

و عندما تنتهي من الكتابة عليك إغلاق الملف:

```
>>> fout.close()
```

14.3 مؤثّر تغيير الصيغة

قرينة write يجب أن تكون محارف، لذلك إن أردنا قيم أخرى في الملف علينا تحويلها إلى محارف. أسهل طريق لذلك هي :str

```
>>> x = 52
>>> fout.write(str(x))
```

الطريقة البديلة هي مؤثر الصيغة %، عند استخدامه على أعداد صحيحة فإنه يكون مؤثر مودولوس (القسمة بدون باق)، لكن إن كان العامل الأول محارف يصبح المؤثر % مؤثر تغيير الصيغة.

العامل الاول هو محارف الصيغة، و الذي يحتوي على واحد أو أكثر من تسلسلات الصيغ التي ستحدد كيف سيصاغ العامل الثاني, و النتيجة تكون محارف.

مثلا، تسلسل الصيغة %d يعني أن العامل الثاني سيصاغ كعدد صحيح (d اختصار لـ decimal) :

```
>>> camels = 42
>>> '%d' % camels
'42'
```

النتيجة '42' هي محارف و يجب عدم الخلط بينها و بين العدد الصحيح 42.

لتسلسل الصيغة أن يظهر في أي مكان في المحارف، فبإمكانك تضمين قيمة في الجملة:

```
>>> camels = 42
>>> 'I have spotted %d camels' % camels
'I have spotted 42 camels
```

و إن كان هناك أكثر من تسلسل صيغة في المحارف، فعلى القرينة الثانية أن تكون توبل، بالترتيب.

المثال التالي يستخدم '%d' ليصيغ عدد صحيح ، '%g' لصياغة رقم حقيقي (و لا تسألني لماذا) و يستخدم كذلك '%s' لصياغة محارف:

```
>>> 'In %d years I have spotted %g %s' % (3, 01, 'camels')
'In 3 years I have spotted 01 camels'
```

يجب أن يتساوى عدد عناصر التوبل مع عدد تسلسلات الصيغ في المحارف. و كذلك على أنماط العناصر أن تكون كما في تسلسلات الصيغ:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

في المثال الاول لم يكن هناك عناصر بما فيه الكفاية، و في الثاني كانت العناصر من النمط الخطأ.

مؤثر الصياغة له قدرات، لكنه صعب الاستخدام، بإمكانك القراءة أكثر عنه هنا

.http://docs.python.org/2/library/stdtypeshtml#string-ormatting

14.4 أسماء الملفات و المسارات

تنظم الملفات في مجلدات (أو دليل) و لكل برنامج شغال "مجلد حالي" ، و هو المجلد الافتراضي لمعظم العمليات مثلا عندما تفتح ملف للقراءة فسيبحث عنه بايثون في المجلد الحالي.

يوفر مديول os اقترانات للعمل مع الملفات و المجلدات (os من operating system) و os.get ترجع اسم المجلد الحالي:

```
>>> import os
>>> cwd = osget.cwd()
>>> print cwd
/home/dinsdale
```

cwd هي اختصار لـ المجلد الشغال الحالي (current working directory). النتيجة في هذا البرنامج هي home/dinsdale/ و التي هي المجلد القاعدة لمستخدم اسمه dinsdale.

و محارف كـ cwd و التي تتعرف على الملف تسمى مسار. و المسار النسبي يبدأ من المجلد الحالي، المسار المطلق يبدأ من من أعلى مجلد في نظام الملفات.

كانت المسارات التي رأيناها حتى الان اسماء ملفات بسيطة، لذلك فهي تُنسَب للمجلد الحالي. لكي تجد المسار المطلق لملف يمكنك استخدام os.path.abspath:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memotxt'
```

تفحص open.path.exists إذا ما كان الملف موجودا:

```
>>> os.path.exists('memo.txt')
True
```

و إن كان موجودا ستفحص os.path.isdir إذا ما كان مجلد:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

أما os.path.isfile فستفحص إذاما كان ملفا.

ترجع os.listdir قائمة بالملفات (و المجلدات الاخرى) من المجلد المعطى:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

و لاستعراض هذه الاقترانات، "يطوف" المثال التالي في مجلد، و يطبع أسماء الملفات كلها، و ينادي نفسه اجتراريا على كل المجلدات

```
def walk(dirname):
   for name in os.listdir(dirname):
      path = os.path.join(dirname, name)
      if ospathisfile(path):
         print path
      else:
         walk(path)
```

يأخذ os.path.join اسم المجلد و اسم الملف و يضمها ليصبحا مسارا كاملا.

تمرين 14.1 يوفر المديول os اقترانا اسمه walk، و هو شبيه بالطواف في المثال السابق لكنه متنوع أكثر. اقرأ وثائقه ثم استخدمه لطباعة أسماء الملفات في مجلد و اسماء المجلدات الفرعية.

الحل: `http://thinkpython.com/code/walk.py`.

14.5 التقاط الاستثناءات

كثيرة هي الامور التي قد تخفق عندما تحاول قراءة أو كتابة الملفات. فإن حاولت فتح ملف غير موجود ستحصل على خطأ IOError:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

و ان كنت لا تملك الحق في الوصول إلى ملف:

```
>>> fout = open('/etc/passwd', 'w')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

و إن حاولت فتح مجلد للقراءة ستحصل على:

```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

لتجنب كل هذه الأخطاء يمكنك استخدام اقترانات مثل os.path.exists و os.path.isfile. لا أن فحص كل الاحتمالات سيستغرق الكثير من الوقت و من السطور البرمجية (إن نبهك الخطأ Errno 21 إلى شيء فليكن أن هناك 21 شيئا قد يخفق).

من الافضل أن تمضي قدما و **تحاول** – ثم تتعامل مع المشاكل عند حدوثها – و هو بالضبط ما تقوم به عبارة try. نحو try شبيه بنحو عبارة if:

```
try:
   fin = open('bad_file')
   for line in fin:
      print line
      fin.close()
except:
   print 'Something went wrong'
```

يبدأ بايثون بتنفيذ فقرة try، و إن كان كل شيء على ما يرام سيتخطى فقرة except، أما إن حصل استثناء و سيقفز خارج فقرة try و ينفذ فقرة except.

تداول الأخطاءباستخدام try يسمى إلتقاط catching الاستثناء exception. في هذا المثال تطبع فقرة except رسالة خطأ لكنها ليست ذات دلالة كبيرة. في العموم فإن التقاط الاستثناءات يسمح لك بإصلاح المشكلة، أو أن تحاول ثانية، أو على الاقل أن توقف البرنامج بلباقة.

تمرين 14.2 اكتب اقترانا اسمه sed يأخذ كقرائن: محارف نموذجية و محارف بديلة و اسمين لملفين. يجب أن يقرأ الملف الاول و يكتب محتوياته في الملف الثاني (ينشئ ملفا ثانيا إن تطلب الامر). إن وُجدت المحارف النموذجية في أي مكان في الملف فيجب أن تستبدل بالمحارف البديلة.

إن ظهر خطأ خلال فتح أو قراءة أو إغلاق الملفات، يجب على برنامجك التقاط الاستثناء، ثم يطبع رسالة وجود خطأ و يخرج من البرنامج الحل: `http://thinkpython.com/code/sed.py`.

14.6 قواعد البيانات

قاعدة البيانات هي ملف مجهز لكي يخزن البيانات. معظم قواعد البيانات منظمة كقاموس، من ناحية أنها تخط من المفاتيح إلى القيم. الفرق الاكبر بينها أن قاعدة البيانات موجودة على قرص (أو أي وسيلة تخزين دائمة) فهي بذلك ثابتة.

يوفر المديول anybm واجهة لإنشاء و تعديل ملفات قواعدالبيانات. و كمثال، سأنشئ قاعدة بيانات تحتوي على اقتباسات لملفات صور.

فتح قاعدة بيانات كفتح أي ملف اخر:

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

النسق c يعني أنه يجب انشاء قاعدة البيانات إن لم تكن منشأة أصلا. و النتيجة هي قاعدة بيانات يمكن استخدامها (لمعظم العمليات) كما يستخدم القاموس. فإن أوجدت عنصرا جديدا فسيُحَدِّث anybm قاعدة البيانات:

```
>>> db['cleesepng'] = 'Photo of John Cleese'
```

و إن قمت بتعيين آخر لأي من المفاتيح فسيبدل anybm القيمة القديمة:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk'
>>> print db['cleesepng']
Photo of John Cleese doing a silly walk
```

الكثير من طرق القواميس تعمل مع كائنات قواعد البيانات. و يعمل معها أيضا التكرار بواسطة عبارة for:

```
for key in db:
print key
```

و كما هو الحال مع الملفات الاخرى، عليك اغلاق قاعدة البيانات بعدما تنتهي من عملك:

```
>>> db.close()
```

14.7 التخليل Pickling

هنالك تقييد لـ anydbm. و هو أن كلّا من المفاتيح و القيم يجب أن تكون محارف. و ستحصل على خطأ إن حاولت استخدام أي نمط اخر.

هنا سيساعدنا مديول التخليل pickle فهو يترجم أي نوع من الكائنات تقريبا إلى محارف مناسبة للتخزين في قاعدة بيانات، ثم يترجم المحارف إلى كائنات مرة أخرى كما كانت.

يأخذ pickle.dumps كائنا كبرمتر و يرجع محارف تمثله (dump**s** هي اختصار **dump string** أفرغ المحارف في..) :

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na'
```

الصيغة هنا ليست واضحة للقراءة بالنسبة للبشر، لأن المقصود هو تسهيل تأويلها من قبل pickle.

pickle.loads (أيضا **load string**) يعيد بناء الكائن:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> print t2
[1, 2, 3]
```

و رغم أن الكائن الجديد له نفس قيمة القديم، إلا أنه (في العموم) ليس القديم:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

بكلمات اخرى فالتخليل و إزالة التخليل (الطوزجة؟) لها نفس تأثير نسخ الكائنات.

يمكنك استخدام pickle لتخزين غير المحارف في قاعدة البيانات، و هو أمر شائع لدرجة أنه كُبسِل في مديول يسمى shelve.

تمرين 14.3 إن كنت قد حملت حلي للتمرين 12.4 من

http://thinkpython.com/code/anagram_sets.py.

سترى بإنه ينشئ قاموسا يصل ما بين محارف (من الحروف) و قائمة الكلمات التي يمكن تهجئتها بحروف المحارف تلك فمثلا opst موصولة بالقائمة ['opts' , 'post' , 'pots' , 'spot' , 'tops'].

اكتب مديول يستورد anagram_sets و يزودنا باقترانين store_anagrams الذي يخزن قاموس الجناس التصحيفي في "رف" و read_anagrams الذي يبحث عن كلمة و يرجع قائمة بكل جناساتها. الحل: http://thinkpython.com/code/anagram_db.py

14.8 الأنابيب Pipes

توفر معظم أنظمة التشغيل واجهة لسطر الاوامر، تعرف أيضا بالصدفة Shell. الصدفات في العادة توفر أوامر للتنقل في نظام الملفات، و إطلاق التطبيقات. مثلا في يونكس يمكن تغيير المجلد باستخدام cd و عرض محتوياته بـ ls و اطلاق مستعرض ويب بطباعة (مثلا) firefox.

فأي برنامج يمكنك إطلاقه من الصدفة يمكنك إطلاقه من بايثون أيضا باستخدام أنبوب pipe. الانبوب هو كائن يمثل برنامجا شغالا.

مثلا في يونكس، يعرض الأمر ls -l محتويات المجلد الحالي (في صيغة long). يمكنك إطلاق ls بـ os.popen [(1)]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

تكون القرينة محارف تحتوي أمر الصدفة، و تكون قيمة المرتجع كائن يتصرف كما يتصرف الملف المفتوح. يمكنك قراءة المخرجات من عملية ls سطرا بعد سطر بـ readline أو قراءة كل شيء بـ read :

```
>>> res = fp.read()
```

و عندما تنتهي منه تغلقه كما تغلق ملف:

```
>>> stat = fp.close()
>>> print stat
None
```

المرتجع هو اخر حالة لعملية ls ، و None تعني أن العملية انتهت بشكل عادي (بدون أخطاء)

مثلا، معظم أنظمة يونكس بها أمر اسمه md5sum يقرأ محتويات ملف و يحسب "checksum" أي "مجموع اختباري"، يمكنك القراءة عن MD5 من http://en.wikipedia.org/wiki/Md5.

يوفر هذا الامر طريقة فعالة لاختبار إذا ما كان للملفان مختلفان نفس المحتوى. فاحتمال تساوي المجاميع الاختبارية لمحتويين مختلفين ضئيلة لدرجة الاهمال (قد تحدث إن انهار الكون).

يمكنك استخدام أنبوبا لتشغيل md5sum من خلال بايثون و تحصل على النتيجة:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
```

```
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print res
1e0033f0ed0656636de0d75144ba32e0 booktex
>>> print stat
None
```

تمرين 14.4 إن كانت لديك مجموعة ضخمة من ملفات mp3، فقد تكون إحداها أو أكثر نسخ لنفس الاغنية، و لكنها مخزنة باسم اخر أو في مجلد اخر. هدف هذا التمرين هو البحث عن المكررات.

1. اكتب برنامجا يبحث في مجلد و في مجلداته الفرعية و يرجع قائمة بكل الملفات التي ادخلت لاحقتها (مثل `mp3.`) تلميح: العديد من الاقترانات المفيدة في التلاعب بالملفات و أسمائها موجودة في `os.path` .
2. تستطيع استخدام `md5sum` للتعرف على المتكررات عن طريق احتساب ال (checksum) لكل ملف فإن كان للملفين نفس الحسبة فعلى الاغلب هما تكرار لنفس المحتوى.
3. للتأكد من نتيجة الفحص يمكنك استخدام أمر يونكس `diff`.

14.9 كتابة المديولات

كل ملف يحتوي على نص برمجي لبايثون يمكن استيراده كمديول. مثلا إن كان لديك ملف اسمه `wc.py` به النص البرمجي التالي:

```
def linecount(filename):
count = 0
for line in open(filename):
count += 1
return count

print linecount('wc.py')
```

إن شغلت هذا البرنامج فسيقرأ نفسه و يطبع عدد السطور في الملف، و التي هي 7:

```
>>> import wc
7
```

أصبح لديك الان كائن مديول اسمه `linecount` :

```
>>> wc.linecount('wc.py')
7
```

إذن فهكذا تكتب مديولا في بايثون.

المشكلة الوحيدة في هذا المثال هي أنك عندما تستورد هذا المديول فإنه ينفذ النص البرمجي الإختباري في الأسفل. عادة عندما تستورد مديول فإنه يعرِّف اقترانات جديدة، لكنه لا ينفذها.

```
if __name__ == '__main__':
      print linecount('wc.py')
```

أصبحت `popen` غير مرغوبة، أي يجب التوقف عن استخدامها و البدء باستخدام `subprocess`. لكن للحالات البسيطة، أجد أن `subprocess` معقدة أكثر من اللازم. لذلك سأستخدم `popen` حتى يزيلوها.

البرامج التي ستُستورَد كمديول لها في الغالب الصيغة التالية:

__name__ متغير جاهز يُطلَق عند ابتداء البرنامج. فإن كان البرنامج شغالا كنص برمجي، فإن قيمة __name__ هي __main__ ، و في هذه الحالة سينفذ نص الاختبار. و إن لم يكن، أي إن كان مديولا يتم استيراده، فسيتخطى نص الاختبار.

تمرين 14.5 اطبع هذا المثال في ملف اسمه wc.py ، ثم شغله كنص برمجي. ثم شغل مفسِّر بايثون و استورده استيرادا، import wc ، ما هي قيمة __main__ عندما كان المديول مستوردا؟

تحذير: ان استوردت مديولا كنت قد استوردته من قبل، فلن يفعل بايثون أي شيء، فلا يعيد قراءة الملف، حتى و إن عُدِّل عليه.

إن أردت إعادة تحميل الملف عليك استخدام الاقتران الجاهز reload ، الا أنه مخادع بعض الشيء، لذا فآمن شيء هو إعادة بدء المفسِّر ثم استيراد المديول من جديد.

14.10 علاج الأخطاء

عند قراءة و كتابة الملفات قد تقع في مشاكل مع المساحات البيضاء. هذه الأخطاء صعبة الاكتشاف لأن مسافات الجدولة و السطور الجديدة في الغالب غير مرئية:

```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2 3
4
```

هنا قد يساعدك الاقتران الجاهز repr فهو يأخذ أي كائن كقرينة و يرجع محارف تمثله. و للمحارف، تمثل المساحات البيضاء بتسلسلات من الخطوط المائلة

```
>>> print repr(s)
'1 2\t 3\n 4'
```

التي قد تساعد في علاج الاخطاء.

مشكلة اخرى قد تقع فيها هي أن أنظمة التشغيل المختلفة تستخدم حروفا مختلفة للتعبير عن سطر جديد. فمنها ما يستخدم "سطر جديد newline" و تكتب \n ، و منها ما يستخدم "return رجوع" و تكتب \r و منها ما يستخدم كليهما. فإن نقلت الملفات بين هذه الانظمة الغير متوافقة فقد تتسبب بالمشاكل.

توجد تطبيقات في معظم الانظمة تحول من صيغة إلى اخرى بإمكانك العثور عليها (و القراءة عنها) على http://en.wikipedia.org/wiki/Newline.

طبعا يمكنك كتابة التطبيق الخاص بك.

14.11 المعاني

ثابت persistant: تتعلق بالبرنامج الذي يستمر في العمل إلى ما لا نهاية و يحتفظ على الاقل بجزء من بياناته.

مؤثر الصيغ format operator: مؤثر % يأخذ محارف الصيغة و توبل و يولد محارف تحتوي على عناصر التوبل مصاغة حسب ما تحدده محارف الصيغة.

محارف الصيغة format string: محارف تستخدم مع مؤثر الصيغة و تحتوي على تسلسل الصيغة.

تسلسل الصيغة format sequence: تسلسل من الحروف في محارف الصيغة ، مثل `%d` يحدد كيفية صياغة قيمة ما.

ملف نصي text file: تسلسل من الحروف مخزن بشكل دائم على قرص صلب.

مجلد directory: مجموعة من الملفات تحت اسم واحد.

مسار (مساق) path: محارف تحدد الملف.

مسار نسبي relative path: مسار يبدأ من المجلد الحالي.

مسار مطلق absolute path: مسار يبدأ في أعلى موقع في نظام الملفات.

التقاط catch: منع الاستثناء من إنهاء البرنامج باستخدام عبارتي `try` و `except` .

قاعدة بيانات database: ملف تنظم محتوياته على شكل مجلد ترتبط فيه المفاتيح بالقيم المقابلة.

14.12 تمارين

تمرين 146 هناك طرق في مديول `urllib` للتعامل مع ال *URL* و تحميل المعلومات من الويب. المثال التالي يحمل و يطبع رسالة نصية من `thinkpython.com`:

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for line in conn:
   print line.strip()
```

شغل هذا النص و اتبع التعليمات التي سيظهرها. الحل:

`http://thinkpython.com/code/zip_code.py`.

الفصل الخامس عشر

الفئات و الكائنات

تتوفرة أمثلة على نصوص البرمجة لهذا الفصل على http://thinkpython.com/code/Point1.py.

و حلول التمارين موجودة على http://thinkpython.com/code/Point1_soln.py.

15.1 أنماط عرّفها المستخدم User-defined types

لقد استخدمنا الكثير من انماط بايثون الجاهزة، لقد حان الوقت لتعريف أنماط جديدة. و على سبيل المثال سننشئ نمطا اسمه Point سيمثل نقطة في فراغ ثنائي الابعاد.

في عرف الرياضيات تكتب النقاط بين قوسين و تفصل بين إحداثياتها فاصلة. مثلا (0,0) تمثل نقطة الاصل، و (س,ص) تمثل النقطة التي تبعد س وحدات إلى اليمين و ص وحدات إلى الأعلى من نقطة الاصل.

هنالك العديد الطرق التي يمكننا تمثيل النقاط بها في بايثون:

- يمكننا تخزين الإحداثيات بشكل منفصل في متغيرين س و ص.
- يمكننا تخزينها كعناصر في قائمة أو توبل.
- يمكننا إنشاء نمط جديد يمثل النقاط ككائنات.

إنشاء نمط جديد معقد قليلا إذا ما قورن بالخيارين الاخريين، الا أن له حسنات ستتبدى لك بسرعة.

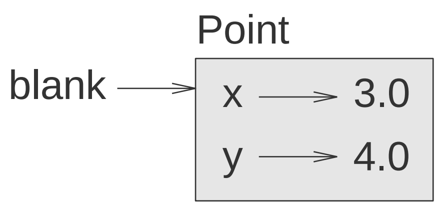
النمط الذي يعرّفِه المستخدم يسمى فئة Class. و تعريف الفئة يكون كالتالي:

```
class Point(object):
    """Represents a point in 2-D space"""
```

تشير الترويسة إلى أن الفئة الجديد هي نقطة، و التي هي عبارة عن object ما، و الذي هو نمط جاهز.

المتن عبارة عن محارف للتوثيق، تشرح الهدف من الفئة. يمكنك تعريف اقترانات و متغيرات داخل الفئة، سنعود لهذا فيما بعد.

و بتعريف فئة اسمها Point سيُخلق كائن فئة



الشكل 15.1: رسم للكائن

```
>>> print Point
<class '__main__Point'>
```

و لأن تعريف `Point` كان في المستوى الاعلى فإن "اسمها الكامل" يكون: `__main__.Point`.

كائن الفئة كالمصنع الذي ينتج الكائنات. و لتنتج نقطة ستنادي `Point` كأنها اقتران:

```
>>> blank = Point()
>>> print blank
<__main__Point instance at 0xb7e9d3ac>
```

و القيمة المرجعة هي مرجع لكائن Point، الذي نعينه لـ `blank`. يسمى إيجاد كائن جديد بالتجلية instantiation، و الكائن المنشأ يسمى instance of the class تجلية للفئة.

و عندما تطبع تجلية فسيخبرك بايثون إلى أي فئة تنتمي و في أي مكان خزنت في الذاكرة (البادئة 0x تعني أن ما يليها هو رقم سداسي عشري).

15.2 الخصال :Attributes

يمكنك تعيين قيم للتجلية باستخدام التنويت بالنقاط dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

يشبه هذا النحو النحو الذي يستخدم لاختيار العناصر من مديول، مثل `mathpi` أو `string.whitespace` مع أننا في هذه الحالة نعين قيما لعناصر مسماة في الكائن. هذه العناصر تسمى خصال **attributes**.

الكلمة attribute هي اسم، و التاء فيها مشددة خلافا للفعل atribute .

يبين الرسم التالي نتيجة هذه التعيينات. رسم الحالة الذي يعرض الكائن و خصاله يسمى رسم الكائن object diagram أنظر الشكل 15.1.

مرجع الكائن Point هو المتغير `blank` و يحتوي هذا الكائن على و هو يحتوي على خصلتين كل منها تشير إلى عدد حقيقي.

يمكنك قراءة قيمة الخصلة باستخدام نفس النحو:

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

التعبير `blank.x` يعني "اذهب إلى الكائن الذي مرجعه `blank` و أحضر قيمة `x`".

نحن في هذه الحالة نعين تلك القيمة للمتغير `x` و لا يوجد تعارض بين المتغير `x` و الخصلة `x`.

يمكنك استخدام النتوتة كجزء من أي تعبير, مثلا:

```
>>> print '(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print distance
5.0
```

يمكنك تمرير تجلّية كقرينة بالطريقة المعتادة، مثلا:

```
def print_point(p):
   print '(%g, %g)' % (p.x, p.y)
```

هنا print_point تأخذ نقطة كقرينة و تعرضها بنتوتة رياضية> و لاستدعائها يمكنك تمرير blank كقرينة:

```
>>> print_point(blank)
(3.0, 4.0)
```

في داخل الاقتران p هي اسم مرجع متعدد من plank، فإن عدل الاقتران على p ستتغير blank.

تمرين 15.1 اكتب اقترانا اسمه distance_between_points يأخذ نقطتين كقرائن و يرجع المسافة بينهما.

15.3 المستطيلات

أحيانا يكون من الواضح ماذا يجب أن تكون خصال كائن ما، لكن في أحيان أخرى عليك اتخاذ القرارات. فتخيل مثلا أنك تصمم فئة لتمثيل المستطيلات. فما هي الخصال التي ستستعملها لتحديد موقع و حجم المستطيل؟ للتسهيل، يمكنك تجاهل الزوايا، فالمستطيل إما أفقي أو عمودي.

هناك احتمالان على الاقل:

- يمكنك تحديد ركن المستطيل (أو حتى مركزه) و العرض و الطول .
- يمكنك تحديد ركنان متقابلان.

حتى هذه اللحظة لا نستطيع التفضيل بين الاحتمالين، لذلك سنطبق الخيار الاول، كمثال فقط:

هذا هو تعريف الاقتران:

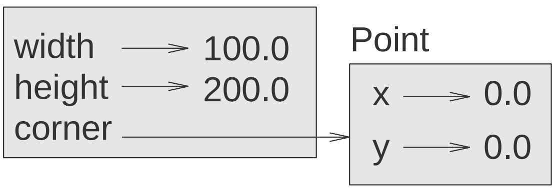
```
class Rectangle(object):
   """Represents a rectangle

   attributes: width, height, corner
   """
```

يُحدِّد نص التوثيق الخصال: width و height هما عددان و corner هي كائن نقطة يحدد الركن السفلي الأيسر.

و لتمثيل مستطيل عليك أنشاء تجلية لكائن المستطيل ثم تعين قيم للخصال:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```



الشكل 152 رسم الكائن

```
Box.corner = Point()
Box.corner.x = 0.0
Box.cornery = 00
```

التعبير box.corner.x يعني "اذهب إلى الكائن الذي مرجعه box و اختر من هناك الخصلة التي اسمها corner ثم اذهب إلى ذلك الكائن و اختر الخصلة التي اسمها x."

يبين الشكل 15.2 حالة هذا الكائن، و هي حالة كائن عندما يكون خصلة لكائن اخر. تسمى هذه الحالة embedded مضَمَّن.

15.4 التجليات كقيم مرتجعة

يمكن للاقترانات أن ترجع تجليات. فمثلا find_center تأخذ Rectangle كقرينة و ترجع Point تحتوي على إحداثيات مركز المستطيل:

```
def find_center(rect):
   p = Point()
   p.x = rect.corner.x + rect.width/2.0
   p.y = rect.corner.y + rect.height/2.0
   return p
```

في المثال التالي يمرر box كقرينة و تعين النتيجة Point إلى center :

```
>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)
```

15.5 الكائنات ليست ثبيتة

يمكنك تغيير حالة الكائن بعمل تعيين لإحدى خصاله. فمثلا لتغيير حجم المستطيل بدون تغيير موقعه، يمكنك تعديل قيم width و height:

```
Box.width = box.width + 50
Box.height = box.width + 100
```

و يمكنك أيضا كتابة اقترانات تعدل على الكائنات. مثلا grow_rectangle تأخذ كائن المستطيل و رقمين dwidth و dheight و تضيف هذه الارقام لعرض و طول المستطيل:

```
def grow_rectangle(rect, dwidth, dheight):
rect.width += dwidth
rect.height += dheight
```

المثال التالي يعرض التأثير:

```
>>> print box.width
100.0
>>> print box.height
2000
>>> grow_rectangle(box, 50, 100)
>>> print box.width
150.0
>>> print box.height
300.0
```

في داخل الاقتران كان rect واحدا من مراجع متعددة لـ box ، فإن عدل الاقتران على rect سيتغير box أيضا

تمرين 15.2 اكتب اقترانا اسمه move_rectangle يأخذ رقمين dx و dy يجب أن يغير موقع المستطيل بإضافة dx إلى الاحداثي x لـ corner و dy إلى الإحداثي y لـ corner.

15.6 النسخ

تعدد المرجعيات قد يجعل قراءة البرامج صعبة، لأن التغيير في مكان ما سيكون له تأثير غير متوقع في مكان اخر. من الصعب ملاحقة كل المتغيرات التي قد تتعلق بذلك الكائن.

نسخ الكائنات عادة ما يكون البديل لتعدد المرجعيات. يحتوي المديول copy على الاقتران copy الذي يمكنه مضاعفة أي كائن:

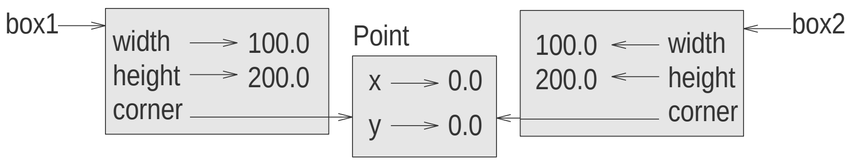
```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> import copy
>>> p2 = copy.copy(p1)
```

يحتوي p1 و p2 على نفس البيانات، لكنهما ليسا نفس النقطة:

```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

المؤثر is يشير إلى أن p1 و p2 ليسا نفس الكائن، و هو ما توقعناه. قد تكون توقعت أيضا أن == ستنتج True لأن هذه الكائنات تحتوي على نفس البيانات. في هذا الحال فقد خاب ظنك، لأنك ستعرف الآن أنه في بعض الحالات يكون تصرف == كتصرف is ، فهذا المؤثر سيفحص هوية الكائن و ليس تساوي الكائن. هذا السلوك يمكن تغييره– سنرى فيما بعد كيف.

إن استخدمت copy.copy لنسخ مستطيل سترى بأنها تنسخ الكائن و ليس النقطة التي يتضمنها.



الشكل 153 رسم الكائن

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

يظهر الشكل 15.3 ما يبدو عليه رسم الكائن. تسمى العملية نسخ ضحل، لإنها تنسخ الكائن و أي مراجع قد يحتويها، لكن لا تنسخ الكائنات المضمنة فيه.

في معظم التطبيقات لن يكون هذا ما تريده. في هذا المثال استدعاء grow_rectangle على أحد المستطيلين لن يؤثر على الاخر، لكن استدعاء move_rectangle سيؤثر على الاخر! و هذا السلوك مشوش و مرتع للخطأ.

لكن لحسن الحظ فإن لدى المديول copy طريقة اسمها deepcopy و هي لا تنسخ الكائن فقط بل الكائنات التي يشير إليها أيضا، و الكائنات التي تشير إليها، و هكذا لعلك لن تفاجأ من تسمية هذه العملية بالنسخ العميق.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

الكائنان box3 و box مختلفان تماما.

تمرين 15.3 اكتب نسخة من move_rectangle تخلق و ترجع مستطيلا جديدا بدلا من التعديل على القديم.

15.7 علاج الأخطاء

عند ابتدائك العمل على الكائنات ستصادفك استثناءات جديدة، إن حاولت مثلا الوصول إلى خصلة غير موجودة ستحصل على AttributeError:

```
>>> p = Point()
>>> print p.z
AttributeError: Point instance has no attribute 'z'
```

إن لم تكن متأكدا من نمط الكائن فيمكنك السؤال:

```
>>> type(p)
<type '__main__Point'>
```

و إن لم تكن متأكدا إن كان للكائن خصلة معينة فيمكنك استخدام الاقتران الجاهز hasattr :

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

للقرينة الاولى أن تكون أي كائن، و تكون القرينة الثانية محارف تحتوي على اسم الخصلة.

15.8 المعاني

فئة Class : نمط يعرفه المستخدم. تعريف الفئة ينشئ كائن فئة جديد.

كائن فئة class object: كائن يحتوي على معلومات عن نمط عرفه المستخدم.

تجلِّية instance : كائن ينتمي إلى فئة.

خصلة attribute : أحد القيم المسماة المرتبطة بكائن.

(كائن) متضمن (embedded (object: كائن مخزن كخصلة داخل كائن اخر.

نسخ ضحل shallow copy : أن تنسخ محتويات كائن بما فيها أي مراجع لكائنات مضمنة، و يستخدم فيها الاقتران copy من المديول copy.

نسخ عميق deep copy : أن تنسخ محتويات كائن و تنسخ أيضا الكائنات المضمنة فيه، و الكائنات المضمنة فيهاو هكذا، يستخدم فيها الاقتران deepcopy من المديول copy .

رسم الكائن object diagram : رسم يظهر الكائنات و خصالها و قيم خصالها.

15.9 تمارين

تمرين 15.4 يوفر سومبي Swampy (انظر الفصل 4) مديولا اسمه World ، و الذي يعرِّف نمطا عرَّفه المستخدم اسمه World و يمكنك استيراده كالتالي:

```
from swampy.World import World
```

أو، حسبما نصبت سومبي، قد يكون هكذا:

```
from World import World
```

النص التالي ينشئ كائن World ثم ينادي على طريقة mainloop التي تنتظر المستخدم.

```
world = World()
world.mainloop()
```

ستظهر نافذة عليها شريط العنوان و مربع فارغ. سنستخدم هذا المربع لرسم النقاط و المستطيلات و الاشكال الاخرى أضف السطور التالية قبل نداء mainloop ثم شغل البرنامج مرة أخرى.

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

يجب أن ترى الان مستطيلا أخضرا بحدود سوداء< السطر الاول سيوجِد قماشة الرسم، و التي ستظهر كمربع أبيض. لكائن القماشة طرق مثل rectangle لرسم الاشكال المختلفة.

bbox هي قائمة قوائم تمثل "صندوق التحديد"للمستطيل. أول زوجين من الاحداثيات هما زاوية المستطيل السفلى إلى اليسار، و الزوجين الثانيين للزاوية العليا اليمنى.

بإمكانك رسم دائرة هكذا:

```
Canvas.circle([-25,0], 70, outline=None, fill='red')
```

البرمتر الاول هو زوجي إحداثيات مركز الدائرة، و الثاني هو نصف القطر.

إن أضفت هذا السطر للبرنامج فيجب أن تحصل على العلم الوطني لبنغلادش (أنظر http://en.wikipedia.org/wiki/Gallery_of_sovereign-state_flags).

1. أكتب اقترانا اسمه draw_rectangle يأخذ قماشة و مستطيلا كبرمترات و يرسم تمثيلا للمستطيل على القماشة.
2. أضف خصلة اسمها color لكائنات المستطيل ثم عدل على draw_rectangle بحيث يستخدم خصلة color لتغيير لون المستطيل.
3. أكتب اقترانا اسمه draw_point يأخذ قماشة و نقطة كقرائن ثم يرسم تمثيلا للنقطة على القماشة.
4. عرف فئة جديدا اسمها Circle لها الخصال المناسبة. ثم أنشئ تجليات لكائن Circle . اكتب اقترانا اسمه draw_circle يرسم دوائر على القماشة.
5. اكتب برنامجا يرسم علم جمهورية التشيك تلميح: يمكنك رسم المضلع هكذا:

```
points = [[-150,-100], [150, 100], [150, -100]]
canvas.polygon(points, fill='blue')
```

لقد كتبت برنامجا صغيرا يسرد الالوان المتاحة، تستطيع تحميله من:

http://thinkpython.com/code/color_lis.tpy

الفصل السادس عشر

الفئات و الاقترانات

أمثلة النصوص البرمجية لهذا القسم متوفرة من: http://thinkpython.com/code/Time1.py

16.1 الوقت

سنعرِّف فئة اسمها Time كمثال آخر على الانماط التي يعرفها المستخدم. و هي تسجل الوقت بالنسبة لليوم. تعريف الاقتران يكون كالتالي:

```
class Time(object):
   """Represents the time of day.

   attributes: hour, minute, second
   """
```

ثم يمكننا إنشاء كائن Time جديد و تعيين خصال للساعات و الدقائق و الثواني:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

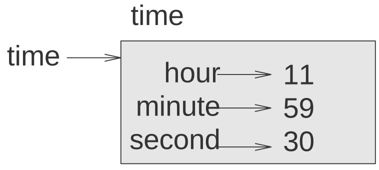
رسم الحالة لكائن Time يبدو كما في الشكل 16.1.

تمرين 16.1 اكتب اقترانا اسمه print_time يأخذ كائن Time و يطبعه على شكل ساعات:دقائق:ثوانيتلميح: يطبع التسلسل '%2d' اقسام الاعددا صحيحا من خانتين على الاقل و يسبقها 0 ان تطلب الامر.

تمرين 16.2 اكتب اقترانا بوليانيا اسمه is_safer يأخذ كائني Time ، هما t1 و t2 و يرجع True إن كان t1 يأتي بعد t2 زمنيا ، و إلافسيرجع False .

16.2 الاقترانات البحتة

سنكتب في الاقسام القليلة القادمة اقترانين لجمع قيم الوقت. و هما يعرضان نوعين من الاقترانات: الاقترانات البحتة و المعدِّلات. و هما يشرحان أيضا خطة تطوير أُسمّيها النموذج المصغر و الترقيع، Prototype and patch. و هي طريقة للتعامل مع المشاكل المعقدة بعمل نموذج مصغر، و منثم التعامل مع التعقيدات كلما تطورت.



الشكل 16.1: رسم الكائن

هنا نموذج مصغر من add_time:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

ينشئ الاقتران كائن Time جديد، و يهيء خصاله ثم يرجع مرجعا لهذا لكائن الجديد. يسمى هذا اقتران بحت لأنه لا يعدل على أي من الكائنات التي مررت إليه كقرائن. و ليس له تأثير، كعرض قيمة أو الحصول على مدخل من المستخدم، غير إرجاع قيمة.

من أجل فحص هذا الاقتران، سأنشئ كائني Time اثنين: start و الذي يحتوي على وقت البدء لفلم، مثل "مونتي بايثون و الكأس المقدسة" ، و duration الذي يحتوي على زمن تشغيل الفلم، و الذي هو ساعة و 35 دقيقة.

ثم سيعلم add_time متى سينتهي عرض الفلم.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

قد لا تكون النتيجة 10:80:00 ما تأملته. اشكلة الاقتران أنه لا يتعامل مع الحالات التي تكون مجاميع الثواني أو الدقائق أكبر من ستين. عندما يحصل هذا فإن علينا "حمل" الثواني الزائدة إلى عمود الدقائق أو الدقائق الزائدة إلى عمود الساعات.

هاك نسخة محسنة:

```
def add_time(t1, t2):
    sum = Time()
    sumhour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

رغم أن هذا الاقتران صحيح، إلا أنه ابتدأ بالتضخم سنرى بدائل أقصر فيما بعد.

16.3 المعدِّلات

من المفيد للإقتران أحيانا أن يعدل على الكائنات التي تمرر إليه كبرمترات. و في هذه الحالة تكون التغييرات مرئية بالنسبة للمنادي. الاقترانات التي تعمل هكذا تسمى معدلات.

فـ increment الذي يضيف عدد معطىً من الثواني إلى كائن Time ، يمكن كتابته طبيعيا كمُعدِّل. اليك مسودة سريعة:

```
def increment(time, seconds):
   time.second += seconds

   if time.second >= 60:
      time.second -= 60
      time.minute += 1

   if time.minute >= 60:
      time.minute -= 60
      time.hour += 1
```

يقوم السطر الاول بالعملية الاساسية، و الباقي يتعامل مع الحالات الخاصة التي رأيناها من قبل.

أهذا اقتران صحيح؟ ما الذي سيحدث ان كان البرمتر seconds أكبر من ستين؟

لا يكفي في هذه الحالة أن يكون الحمل مرة واحدة، فيجب أن نكرر الحمل إلى أن تصبح قيمة seconds أقل من ستين. أحد الحلول هو استبدال عبارة for بعبارة while سيجعل هذا التعديل الاقتران صحيحا لكن ليس كفؤا جدا

تمرين 16.3 اكتب نسخة صحيحة من increment لا تحتوي على أي حلقة.

يمكن للإقتران البحت أن يقوم بأي شيء يقوم به المعدِّل. بل أن بعض لغات البرمجة لا تسمح إلا بالاقترانات البحتة. و هناك بعض الادلة بأن البرامج التي تحتوي على اقترانات بحتة تُطَوَّر أسرع و بأخطاء أقل من تلك التي تستخدم المعدلات. لكن المعدلات مناسبة للوقت، و برامج الاقترانات أقرب إلى أن تكون أقل كفاية.

في العموم أنصحك باستخدام الاقترانات البحتة طالما كان استعمالها مبررا. و الالتجاء إلى المعدلات فقط إن كانت الحسنات مغرية. هذه المقاربة تسمى أسلوب البرمجة الوظائفي.

تمرين 16.4 اكتب نسخة بحتة من increment تنشئ و ترجع كائن Time جديد بدلا من تعديل البرمترات.

16.4 النماذج المصغرة مقابل التخطيط

اسم خطة التطوير التي أعرضها هو "النماذج المصغرة و الترقيع". لكل اقتران، كتبت نموذج مصغر يقوم بالحسبة الأساسية، ثم اختبرته، و كنت أرقع الأخطاء كلما ظهرت في طريقي.

قد يكون هذا التقرب فعالا إن لم يكن لديك الفهم العميق للمشكلة. لكن التصحيح التدريجي عن طريق الزيادة قد يولد برنامجا معقدا بلا داعي (حيث أنه يتعامل مع العديد من الحالات الخاصة). و أيضا لا يعتمد عليه (خصوصا أنك لن تتأكد أنك عثرت على كل الاخطاء).

البديل هو التطوير المخطط له، و فيه تُسهِّل البصيرة البرمجة كثيرا. في حالتنا هذه البصيرة هي أن كائن Time هو في الحقيقة رقم من ثلاثة خانات قاعدته 60 أنظر http://en.wikipedia.org/wiki/Sexagesimal!

خصلة seconds هي خانة الاحاد و خصلة minutes هي خانة "الستينات" و خصلة الساعات هي خانة "الست و ثلاثين مئات!"

عندما كتبنا add_time و increment كنا نقوم بعملية الجمع على القاعدة 60، و لأجل ذلك كنا نقوم بالحمل من عمود إلى العمود الاعلى.

أما ما لاحظناه الآن فهو يقترح علينا مقاربة اخرى للمشكلة ككل: بوسعنا تحويل كائنات Time إلى أعداد صحيحة و استغلال كون الحاسوب يعرف كيف يقوم بالعمليات الحسابية على الاعداد الصحيحة.

إليك اقترانا يحول الأوقات (كائنات Time) إلى أعداد صحيحة:

```
def time_to_int(time):
   minutes = time.hour * 60 + time.minute
   seconds = minutes * 60 + time.second
   return seconds
```

و هنا اقتران يحول الاعداد الصحيحة إلى أوقات (تذكر بأن divmod تقسم القرينة الاولى على الثانية و ترجع الناتج و الباقي كتوبل):

```
def int_to_time(seconds):
   time = Time()
   minutes, time.second = divmod(seconds, 60)
   time.hour, time.minute = divmod(minutes, 60)
   return time
```

لكي تقنع نفسك بأن هذه الاقترانات صحيحة،قد يتطلب الامر بعض التفكير و القيام ببعض الاختبارات. هناك طريقة لفحصها! اختبر إذا ما كان: time_to_int(int_to_time(x))==x لعدة قيم. كان هذا مثالا على فحص التناسق.

و بمجرد اقتناعك بأنها صحيحة استخدمها لإعادة كتابة add_time :

```
def add_time(t1, t2):
   seconds = time_to_int(t1) + time_to_int(t2)
   return int_to_time(seconds)
```

و هذه النسخة أصغر و أسهل للتأكد من الاولى.

تمرين 16.5 أعد كتابة increment باستخدام time_to_int و int_to_time .

يكون التحويل أحيانا من القاعدة 60 إلى القاعدة 10 أصعب من التعامل مع الوقت تحويل العد من قاعدة لأخرى عسير على الفهم، أفضل حدسنا في التعامل مع قيم الوقت.

لكن إن كانت لدينا البصيرة من البداية للتعامل مع الوقت كأعداد قاعدتها ستين. و استثمرناها في كتابة اقترانات التحويل (time_to_int و int_to_time). لكنا قد حصلنا على برنامج أقصر، و أسهل للقراءة و علاج الأخطاء، و يعول عليه.

أيضا، من السهل إضافة مزايا جديدة له مستقبلا مثلا، تخيل ايجاد الفرق بين زمنين للحصول على قيمة ما مضى من الوقت. السذج هم من سيطرحون بالاقتراض، لأن استعمال اقتران التحويل أسهل و على الاغلب أصح.

المفارقة أن تصعيب المشكلة (أو تعميمها) قد يسهلها أحيانا. (لإن هنالك حالات خاصة اقل من و الاقل من فرص الاخطاء).

16.5 علاج الأخطاء

يكون كائن Time مضبوطا إن كانت قيم الدقائق و الثواني بين 0 و 60 (متضمنة الصفر و لكن ليس الستين) و إن كانت قيمة الساعات موجبة. قيم hour و minute يجب أن تكون أعداد صحيحة أما الثواني فيسمح لها بالكسور.

متطلبات كهذه تسمى "لا متغيرة" لأن علينا أن تكون دائما True أو بمعنى اخر، إن لم تكن True فهناك خطأٌ ما.

كتابة نص يفحص اللامتغيرات مفيد في اقتناص الأخطاء و العثور على مسبباتها. فمثلا ليكن لديك اقترانا اسمه valid_time يأخذ كائن Time و يرجع False إن كان يخالف أحد الثوابت:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

ثم في بداية كل اقتران يمكنك اختبار إذا ما كانت القرائن لا تخالف اللا المتغيرات:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

أو يمكنك استخدام عبارة assert التي تفحص اللا المتغير و تقدم استثناءا إن فشل في الفحص:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

المفيد في عبارات assert هي أنها تفرق بين النص الذي يتعامل مع الحالات العادية و النص المكتوب لفحص الاخطاء.

16.6 المعاني

النموذج المصغر و الترقيع prototype and patch: أسلوب في تطوير البرامج، تكتب فيه مسودة سريعة للبرنامج، ثم تشغل و تختبر، و كلما ظهر خطأ يعالج.

التطوير بناءا على خطة planned programming: اسلوب في تطوير البرامج يعتمد على البصيرة الثاقبة في النظر إلى المشاكل و على التخطيط أكثر منه التطوير العصامي أو النموذج المصغر.

اقتران بحت pure function: اقتران لا يعدل على الكائنات التي تمرر له كقرائن. معظم الاقترانات البحتة اقترانات مثمرة.

المعدل modifier: اقتران يعدل على كائن أو أكثر مما مرر إليه كقرائن. معظم المعدلات اقترانات عقيمة.

لا المتغير invariant: شرط يجب أن يصِح دائما قبل تنفيذ البرنامج.

16.7 تمارين

أمثلة نصوص البرمجة لهذا الفصل متوفرة على http://thinkpython.com/code/Time1.py. الحلول لهذه التمارين متوفرة من http://thinkpython.com/code/Time1_soln.py.

تمرين 16.6 اكتب اقترانا اسمه mul_time يأخذ كائن Time و رقم. و يرجع كائن Time جديد يحتوي على حاصل ضرب الوقت الاصلي و الرقم.

ثم استخدم mul_time لكتابة اقتران يأخذ كائن Time يمثل وقت النهاية لسباق، و رقم يمثل المسافة، و يرجع كائن Time جديد يمثل معدل السرعة (الزمن لكل وحدة مسافة).

تمرين 16.7 يوفر مديول datetime كائنين date و time و هما شبيهان بـ Date و Time الذان في هذا الفصل، إلا أنها توفر مجموعة غنية من الطرق و المؤثرات إقرأ وثائقها على http://docs.python.org/2/library/datetime.Html.

1. اكتب برنامجا باستخدام مديول datetime يحصل على التاريخ الحالي و يطبع اسم اليوم.
2. اكتب برنامجا يأخذ تاريخ الميلاد للمستخدم كمدخل، و يطبع عدد الايام و الساعات و الدقائق و الثواني المتبقية حتى عيد ميلاده القادم.
3. يكون للشخصين الذين ولدا في أيام مختلفة يوم يكون فيه سن أحدهما ضعف سن الثاني و هو "يوم الضعف" اكتب برنامجا يأخذ تاريخي ميلاد ثم يحسب لهما "يوم الضعف".
4. لنزد التحدي قليلا، اكتب برنامجا يحسب اليوم الذي يكون فيه أحدهم أكبر بـ n مرة من الاخر.

الفصل السابع عشر

الفئات و الطرق

أمثلة النصوص البرمجية لهذا الفصل موجودة على http://thinkpython.com/code/Time2.py.

17.1 مزايا البرمجة كائنية المنحى

بايثون هي لغة كائنية المنحى، أي أنها توفر مزايا تدعم هذا المنحى في البرمجة.

ليس من السهل تعريف البرمجة كائنية المنحى، الا أننا رأينا بعضا من صفاتها:

- البرامج مبنية من تعريفات لكائنات و لإقترانات و يعبر عن معظم عمليات الحوسبة فيها بعمليات و كائنات.
- كل تعريف لكائن مرتبط بمفهوم ما أو كائن ما في الواقع. و الاقترانات تعمل على ذلك الكائن بطريقة مقرونة بتلك التي تتعامل فيها الكائنات مع بعضها في الواقع.

مثلا، فئة `Time` التي عرفت في الفصل 16 ترتبط بالطريقة التي يسجل فيها الناس الوقت اليومي، و الاقترانات التي عرفناها هناك ترتبط بالاشياء التي يقوم بها الناس بعد معرفتهم الوقت. و بنفس المنطق، فئات `Point` و `Rectangle` ترتبط بالمفهوم الرياضي للنقطة و المستطيل.

حتى هذه اللحظة لم نستغل بعد المزايا التي يوفرها بايثون لدعم البرمجة كائنية المنحى. هذه المزايا ليست ضرورية حرفيا، فكثير منها عبارة عن تراكيب نحوية بديلة للقيام بأشياء قمنا بها بدونها. لكن في العديد من الحالات تكون البدائل أكثر إيجازا و تنقُل لنا بناء البرنامج بشكل أدق.

مثلا في برنامج `Time` لا توجد علاقة واضحة بين تعريف الفئة و تعريف الاقتران الذي يتبعها. و بالقليل من الاختبار سيتبين أن كل اقتران يأخذ على الاقل كائن `Time` واحد كقرينة.

و هذه الملاحظة هي الدافع للعمل مع **الطرائق**، الطريقة هي اقتران مربوط بفئة معينة. لقد رأينا طرائق للمحارف و القوائم و القواميس و التوبل في هذا الفصل سنعرّف **طرائق المستخدم**.

دِلاليا فالطرائق كالاقترانات، ألا أن هناك فرقان نحويان بينهما:

- الطرائق معرفة داخل الفئات لكي تجعل العلاقة بين الفئة و الطريقة مباشرة.
- نحو الاستدعاء لطريقة يختلف عن نحو نداء الاقتران.

في الاقسام القليلة القادمة سنأخذ الاقترانات من الفصول السابقة و نحولها إلى طرائق. و هذا التحويل هو ميكانيكي بحت، و يمكنك القيام به عن طريق اتباع خطوات واضحة، إن أصبح التحويل من شكل لاخر لا يتعبك، فسيمكنك اختيار أفضل الاشكال التي تناسب المهمة التي تقوم بها.

17.2 طباعة الكائنات

عرفنا في الفصل 16 فئة اسمها Time و في التمرين 16.1 كتبت اقترانا اسمه print_time:

```
class Time(object):
   """Represents the time of day"""

def print_time(time):
   print '%2d:%2d:%2d' % (time.hour, time.minute, time.second)
```

و لنداء هذا الاقتران كان عليك تمرير كائن Time كقرينة:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

كل ما يلزمنا عمله لتحويل print_time إلى طريقة هو نقل تعريف الاقتران إلى داخل الفئة. لاحظ التغير في المسافات البادئة:

```
class Time(object):
def print_time(time):
print '%2d:%2d:%2d' % (timehour, timeminute, timesecond)
```

أصبح هناك الان طريقتان لنداء print_time. الاولى (الأقل شيوعا) هي استخدام نحو الاقترانات:

```
>>> Timeprint_time(start)
09:45:00
```

في استخدام التنويت هنا كان Time هو اسم الفئة و print_time هو اسم الطريقة أما start فقد مُررت كبرمتر.

الطريقة الثانية (و هي موجزة أكثر) هي استعمال نحو الطرائق:

```
>>> start.print_time()
09:45:00
```

و في هذا الاستخدام للتنويت كان print_time هو اسم الطريقة (مرة أخرى) و start هو الكائن الذي استدعيت الطريقة عليه، و الذي سيسمى "الموضوع" تماما مثلما أن المبتدأ في جملة اسمية هو ما تبنى لأجله الجملة. فإن موضوع الطريقة هو ما تبنى لأجله الطريقة.

و بداخل الطريقة يعين الموضوع إلى البرمتر الاول، لذلك هنا عينت start إلى time.

و ما أجمع عليه هو أن يسمى أول برمتر في الطريقة self ، لذلك سيكون أقرب إلى الشائع أن تكتب print_time كالتالي:

```
class Time(object):
   def print_time(self):
      print '%2d:%2d:%2d' % (selfhour, selfminute, selfsecond)
```

السبب وراء هذا الاجماع كان استعارة ضمنية:

- نحو النداء للاقتران، print_time(start) يوحي بأن الاقتران هو الفاعل، فهي تقول شيئا كـ "يا print_time إليك هذا الكائن لكي تطبعه".
- تكون الكائنات في البرمجة كائنية المنحى فاعلات , فاستدعاء طريقة مثل startprint_tim() كأنها "يا

start إطبع نفسك".

هذا التغير في وجهة النظر قد يكون مؤدبا، لكن الفائدة العملية منه غير واضحة، ففي الامثلة التي رأيناها حتى الان لا فائدة ترى منه. لكن إبعاد المسئولية عن الاقتران و إلصاقها بالكائن قد يمكننا من كتابة اقترانات متنوعة أحيانا، و تسهل صيانة و إعادة استخدام النصوص البرمجية.

تمرين 17.1 أعد كتابة time_to_int من القسم 16.4 ليصبح طريقة. قد لا يكون من اللائق اعادة كتابة int_to_time إلى طريقة فما هو الكائن الذي ستستدعيه عليه؟

17.3 مثال اخر

هذه نسخة من increment (من القسم 16.3) اعيدت كتابتها كطريقة:

```
# inside class Time:

   def increment(self, seconds):
      seconds += self.time_to_int()
      return int_to_time(seconds)
```

في هذه النسخة يفترض أن time_to_int كُتبت كطريقة، كما في المثال (17.1)، و لاحظ أيضا أن الاقتران بحت و ليس معدِّلا.

و استدعاء increment يكون كالتالي:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

يعيَّن الكائن increment إلى البرمتر self و تعين القرينة 1337 إلى البرمتر الثاني seconds.

قد تكون هذه الالية معقدة خصوصا عند وجود خطأ. فمثلا إن استدعيت increment بقرينتين ستحصل على:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

و رسالة الخطأ هذه محيرة أيضا، لوجود قرينتين بين القوسين. لكن علينا الانتباه بأن الموضوع هو أيضا قرينة، مما يجعل القرائن ثلاثة.

17.4 مثال معقد أكثر

is_after من التمرين 16.2 معقد أكثر قليلا لأنها تأخذ كائني Time كبرمترات. في هذه الحالة تم الاجماع على تسمية البرمتر الاول self و الثاني other :

```
# inside class Time:

   def is_after(self, other):
      return self.time_to_int() > other.time_to_int()
```

و لاستعمال هذه الطريقة عليك استدعاؤها على أحد الكائنين و أن تمرر الاخر كقرينة:

```
>>> end.is_after(start)
True
```

الجميل في هذا النحو هو أنه يُقرأ كاللغة الطبيعية: "end is after start" .

17.5 طريقة init

طريقة init (إختصار لـ initialization تهيئة) هي طريقة فريدة تُستدعى عندما يستهل الكائن. و اسمها الكامل هو __init__ (خطان سفليان ثم init ثم خطان سفليان آخران). طريقة init لكائن Time ستبدو كالتالي:

```
# inside class Time:

   def __init__(self, hour=0, minute=0, second=0):
      self.hour = hour
      self.minute = minute
      self.second = second
```

و من الشائع أن يكون لبرمترات __init__ نفس أسماء الخصال. هذه العبارة:

```
Self.hour = hour
```

تخزن قيمة البرمتر hour كخصلة لـ self.

البرمترات اختيارية، فإن ناديت Time بدون قرائن ستحصل على القيمة الافتراضية:

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

إن أعطيت قرينة واحدة ستتجاوز hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

و إن أُعطِيَت قرينتين ستتجاوز hour و minute:

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

و إن أعطيت ثلاثة قرائن ستتجاوز القيم الافتراضية الثلاثة

تمرين 17.2 إكتب طريقة init للفئة Point تأخذ x و y كبرمترات اختيارية و عينها للخصلتين المقابلتين لها.

17.6 طريقة __str__

طريقة __str__ هي أيضا طريقة خصوصية مثل __init__ ، و المفترض منها إرجاع محارف تمثل الكائن.

فمثلا هذه هي طريقة str للكائن Time:

```
# inside class Time:

   def __str__(self):
      return '%2d:%2d:%2d' % (self.hour, self.minute, self.second)
```

و عندما تطبع كائنا فإن بايثون يستدعي طريقة str:

```
>>> time = Time(9, 45)
>>> print time
09:45:00
```

عندما أكتب فئة جديدة فغالبا ما أبدأ بكتابة __init__ لأنها تساعدني في تهيئة الكائنات، ثم __str__ المفيدة في علاج الاخطاء.

17.7 التحميل الزائد للمؤثرات

مع تعريف طرق خاصة أخرى فإنك تحدد سلوك المؤثرات على انماط المستخدم فمثلا إن عرفت طريقة اسمها __add__ لفئة Time ، فسيمكنك استخدام المؤثر + على كائنات Time.

و هذا ما قد يبدو عليه التعريف:

```
# inside class Time:

   def __add__(self, other):
      seconds = selftime_to_int() + othertime_to_int()
      return int_to_time(seconds)
```

و هنا كيفية استخدامك له:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

عندما تطبق المؤثر + في كائنات Time فإن بايثون يستدعي __add__ و عندما تطبع النتيجة، يستدعي بايثون __str__ إذن فهناك الكثير خلف هذه الكواليس!

تغيير سلوك المؤثر ليعمل على أنماط المستخدم يسمى التحميل الزائد overload لكل مؤثر في بايثون يوجد طريقة خاصة، مثل __add__ لمزيد من التفاصيل أنظر http://doc.spython.org/2/reference/datamodel.html#specialnames.

تمرين 17.4 أكتب طريقة add لفئة Point.

17.8 الإيفاد على أساس النمط

لقد أضفنا في القسم السابق كائني Time، غير أنك قد تود إضافة عدد صحيح أيضا إلى كائن Time. التالي هو نسخة من __add__ تفحص نمط other ثم تستدعي إما add_time أو increment:

```
# inside class Time:

   def __add__(self, other):
      if isinstance(other, Time):
         return self.add_time(other)
      else:
         return self.increment(other)

   def add_time(self, other):
      seconds = self.time_to_int() + other.time_to_int()
      return int_to_time(seconds)

   def increment(self, seconds):
      seconds += self.time_to_int()
      return int_to_time(seconds)
```

يأخذ الاقتران الجاهز isinstance قيمة و كائن فئة، ثم يرجع True إن كانت القيمة هي تجلية للفئة.

إن كانت other هي كائن Time فإن __add__ ستستدعي add_time و إلا فإنها ستفترض بأن البرمتر هو رقم و ستستدعي increment. هذه العملية تسمى **الايفاد على أساس النمط** لإنها توفد العملية الحوسبية إلى عدة طرق بناءا على أنماط القرائن.

هنا مثال يستخدم المؤثر + مع نمطين مختلفين:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
>>> print start + 1337
10:07:17
```

هذا التطبيق للطريقة ليس تبادليا للأسف، فإن كان العامل الاول عدد صحيح ستحصل على:

```
>>> print 1337 + start
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

المشكلة هي أنه بدلا من الطلب إلى كائن Time أن يضيف عددا، فإن بايثون يطلب إلى العدد الصحيح أن يضيف كائن Time و هو لا يعرف كيف يطلبه الا أن هنالك حل ذكي لهذه المشكلة: الطريقة الخصوصية __radd__ و تعني (right-side add) تستدعى هذه الطريقة عندما يكون موقع كائن Time إلى يمين المؤثر +.

و هذا هو تعريفها:

```
# inside class Time:

   def __radd__(self, other):
      return self__add__(other)
```

و تستخدم هكذا:

```
>>> print 1337 + start
10:07:17
```

تمرين 17.5 أكتب طريقة add للنقاط، تعمل على كل من كائن Point و توبل:

- إن كان العامل الثاني Point فعلى الطريقة إرجاع Point جديدة إحداثيها السيني هو مجموع الاحداثيات السينية للعوامل، و كذلك بالنسبة للإحداثيات الصادية.
- إن كان العامل الثاني توبل فعلى الطريقة إضافة العنصر الاول من التوبل إلى الاحداثي السيني و العنصر الثاني إلى الاحداثي الصادي، ثم ترجع Point جديدة مع النتيجة.

17.9 تعدد الاشكال

الايفاد على أساس النمط مفيد عندما يكون ضروريا، لكن (لحسن الحظ) ليس ضروريا دائما. فيمكنك دوما تجنبه عن طريق كتابة اقترانات تعمل بشكل صحيح مع القرائن من مختلف الانماط.

فالكثير من الاقترانات التي كتبناها للمحارف تعمل مع مختلف التسلسلات مثلا في القسم 11.1 استخدمنا histogram لحساب عدد مرات ظهور حرف في كلمة:

```
def histogram(s):
   d = dict()
   for c in s:
      if c not in d:
         d[c] = 1
      else:
         d[c] = d[c]+1
   return d
```

سيعمل هذا الاقتران مع القوائم و التوبل و حتى مع القواميس طالما كانت عناصر s تقطيعية، ، لإمكانية استخدام العناصر كمفاتيح في d:

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

الاقترانات التي تعمل مع مختلف الانماط تسمى متعددة الاشكال **polymorphic**. يمكن لمتعددات الاشكال تسهيل إعادة استخدام النصوص البرمجية. مثلا الاقتران الجاهز sum الذي يجمع عناصر تسلسل، سيعمل طالما كانت عناصر التسلسل تدعم الجمع.

و بما أن كائنات Time تزودنا بطريقة add فستعمل مع sum:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print total
23:01:00
```

إجمالا، إن كانت جميع العمليات داخل اقتران تعمل مع نمط ما، فالاقتران نفسه يعمل مع ذلك النمط.

أفضل أنواع متعددات الاشكال هي تلك تأتي بالصدفة، عندما تكتشف بأن الاقتران الذي كتبته يعمل على نمط لم تفكر فيه أبدا.

17.10 علاج الأخطاء

من الجائز إضافة خصال لكائن في أي مرحلة من مراحل تنفيذ البرنامج، لكن إن كنت من المؤمنين بنظرية النمط، فسيصبح وجود كائنات من نفس النمط لها مجموعات مختلفة من الخصال أمرا مشكوكا فيه لديك. في العادة ينصح بتهيئة جميع خصال الكائن في طريقة init.

إن لم تكن متأكدا من أن كائن ما له تلك الخصلة، فاستخدم الاقتران الجاهز hasattr (انظر القسم 15.7).

سبيل اخر للوصول إلى خصال كائن هي استخدام الخصلة الخصوصية __dict__ و هي عبارة عن قاموس يوفق بين أسماء الخصال (كمحارف) و بين القيم:

```
>>> p = Point(3, 4)
>>> print p__dict__
{'y': 4, 'x': 3}
```

و لعلاج الأخطاء قد يفيدك ابقاء هذا الاقتران في متناول يدك:

```
def print_attributes(obj):
      for attr in obj__dict__:
print attr, getattr(obj, attr)
```

تمر print_attributes بعناصر قاموس الكائن و تطبع كل خصلة و القيمة المرادفة لها.

الاقتران الجاهز getattr يأخذ كائنا و اسم خصلة (كمحارف) و يرجع قيمة الخصلة.

17.11 واجهة المستخدم و التطبيق

إبقاء ناصية البرمجيات ممسوكة هو أحد أهداف البرمجة كائنية المنحى، يعني هذا أن البرنامج سيظل يعمل حتى عندما تتغير أجزاء النظام الاخرى، و أيضا عند تعديل البرنامج لكي يوافق ما يستجد من المتطلبات.

من مفاهيم التصميم التي تساعد على تحقيق هذا الهدف هو ابقاء واجهة المستخدم منفصلة عن التطبيق. و يعني هذا بالنسبة للكائنات، أن تستقل الطرق التي توفرها الفئات عن كيفية تمثيل الخصال.

مثلا، طورنا في هذا الفصل فئة لتمثيل الوقت، الطرق التي توفرها هذه الفئة تتضمن time_to_int و is_after و add_time.

يمكننا تطبيق هذه الطرائق بعدة أساليب. و تفاصيل هذا التطبيق يعتمد على كيفية إظهار الوقت التي نفضلها. كانت خصال كائن Time في هذا الفصل هي hour و minute و second.

كبديل لها، كان بوسعنا استبدال كل هذه الخصال بعدد صحيح واحد يمثل عدد الثواني التي مرت منذ منتصف الليل. سيسهل هذا التطبيق كتابة بعض الطرق مثل is_after لكنه سيصعب الاخرى.

قد تكتشف بعد أن تطلق فئة جديدة أن هناك تطبيقا أفضل. إن كانت هنالك أجزاء أخرى في برنامجك تستخدم هذه الفئة فسيكون تغيير الواجهة استنفاذا للوقت و منبعا للأخطاء.

لكن إن كنت صممت الواجهة بعناية، فستتمكن من تغيير التطبيق بدون الحاجة إلى تغيير الواجهة، مما يعني أنه لا حاجة لتغيير أجزاء أخرى في البرنامج.

مما يعنيه فصل واجهة المستخدم عن التطبيق هو إخفاء المعلومات. فالنصوص البرمجية في المواقع الاخرى من البرنامج (خارج

تعريف الفئة)، يجب أن تستخدم الطرق لقراءة و تعديل حالة الكائن فقط، لا للوصول إلى خصاله.

تمرين 17.6 حمل النص البرمجي لهذا الفصل من http://thinkpython.com/code/Time2.Py.

غير خصال Time لتصبح عددا صحيحا واحدا يمثل الثواني منذ منتصف الليل. ثم عدل الطرق (و كذلك الاقتران int_to_time) ليعمل مع التطبيق الجديد. سيتوجب عليك تعديل النص الاختباري في main و عندما تنتهي يجب أن تكون المخرجات كما كانت في السابق.

الحل: http://thinkpython.com/code/Time2_soln.py.

17.12 المعاني

لغة كائنية المنحى object -oriented language: لغة برمجة توفر مزايا، مثل فئات المستخدم و نحوٍ للطرق، لتسهيل البرمجة كائنية المنحى.

البرمجة كائنية المنحى object-oriented programming: أسلوب في البرمجة تكون فيه البيانات و العمليات عليها منظمة في فئات و طرائق.

طريقة method: اقتران معرف داخل فئة، يستدعى على تجليات لتلك الفئة.

موضوع subject: الكائن الذي تستدعى عليه الطريقة.

التحميل الزائد للمؤثر operator overloading: تغيير سلوك المؤثر (مثل +) بحيث يعمل مع نمط أوجده المستخدم.

الايفاد على أساس النمط type-based dispatch: قالب برمجي يفحص نمط المؤثر و يستدعي الاقترانات المختلفة على الانماط المختلفة.

متعدد الاشكال polymorphic: صفة للإقتران الذي يعمل مع أكثر من نمط.

إخفاء المعلومات information hiding: مبدأ أن واجهة المستخدم التي يوفرها الكائن يجب ألا تعتمد تطبيقه، خصوصا تمثيل خصاله.

17.13 تمارين

تمرين 17.7 هذه التمارين عبارة عن قصة تحذيرية عن أكثر الأخطاء شيوعا و الأصعب في العثور عليها في بايثون. اكتب فئة اسمها kangaroo بها الطرق التالية:

1. طريقة __init__ تهيئ خصلة اسمها pouch_contents إلى قائمة فارغة
2. طريقة اسمها put_in_pouch تأخذ كائنا من أي نمط و تضيفه إلى pouch_content
3. طريقة __str__ ترجع تمثيلا محارفيا للكائن kangaroo و لمحتويات pouch (= كيس)

اختبر نصك بكائني kangaroo معينا إياهما إلى متغيرين اسمهما kanga و roo ، ثم بعدها أضف roo إلى محتويات كيس kanga

حمل http://thinkpython.com/code/BadKangaroo.py.

الذي به حل المسألة السابقة، و به أيضا بقة كبيرة مقيتة! إعثر عليها و صلحها

إن استسلمت يمكنك تحميل شرح المشكلة و حلها من:

.http://thinkpython.com/code/GoodKangaroo.py

تمرين 17.8 يزود مديول بايثون Visual رسوم ثلاثية الابعاد. و هو ليس دائمًا من ضمن تنصيب بايثون، و عليه فقد يتطلب الامر تحميله من مستودع البرامج لديك، و إلا فمن http://vpython.org.

المثال التالي يخلق فضاءا ثلاثي الابعاد طوله 256 مديول و كذلك عرضه و ارتفاعه و يجعل النقطة (128, 128، 128) المركز "*center*"، ثم يرسم كرة زرقاء.

```
from visual import *

scenerange = (256, 256, 256)
scenecenter = (128, 128, 128)


color = (0.1, 0.1, 0.9) # mostly blue
sphere(pos=scenecenter, radius=128, color=color)
```

هنا color عبار عن توبل من RGB ، أي أن عناصر التوبل هي R و G و B، و مستوياتها بين 0.0 و 1.0 (أنظر .http://en.wikipedia.org/wiki/RGB_color_model

إن شغلت هذا النص فيجب أن ترى نافذة بخلفية سوداء و بها كرة زرقاء. و إن جررت الزر الاوسط فستتمكن من تكبير الصورة و تصغيرها، يمكنك أيضا تدوير الكرة بالجر بالزر الايمن، لكنك لن تلحظ الفرق لأن هناك كرة واحدة في هذا الكون.

توجِد الحلقة التالية مكعبا من الكرات:

```
t = range(0, 256, 51)
   for x in t:
      for y in t:
         for z in t:
            pos = x, y, z
            sphere(pos=pos, radius=10, color=color)
```

1. ضع هذا النص في برنامج و تأكد من أنه يعمل
2. عدل البرنامج بحيث يصبح لكل كرة اللون الذي يناسب موقعها في الفراغ RGB إنتبه لكون الاحداثيات هي في المجال 0-255 و RGB هي في المجال 0.0 و 1.0.
3. حمل و استخدم الاقتران read_colors لتنشئ قائمة بالالوان المتاحة في نظام التشغيل لديك، تكون بها أسماء الالوان و قيمها، و لكل لون مسمى. ارسم دائرة في المكان الذي يقابل قيمة RGB.

يمكنك الاطلاع على حلي على http://thinkpython.com/code/color_space.py.

الفصل الثامن عشر

التوريث

سأقدم في هذا الفصل فئات تمثل أوراق اللعب، و شدات و فئات بوكر. إن كنت لا تلعب البوكر فاقرأ عنها هنا http://en.wikipedia.org/wiki/Poker.

لكن ليس عليك أن تتعلمها فسأقول لك ما الذي ستحتاجه منها في هذه التمارين أمثلة النصوص البرمجية لهذا الفصل متوفرة من http://thinkpython.com/code/Card.py.

إن لم تكن على معرفة بأوراق اللعب فبإمكانك الاطلاع على :

http://en.wikipedia.org/wiki/Playing_cards.

18.1 كائنات البطاقات

هنالك إثنتان و خمسون بطاقة لعب في الشدة، و كل منها ينتمي إلى واحد من الاربعة أشكال، و إلى واحدة من الثلاثة عشر مرتبة. الاشكال هي البستوني Spades و الكبة Hearts و الديناري Diamonds و السباتي Clubs (مرتبة تنازليا حسب لعبة بردج). ثم هناك المراتب و هي: الأص 2-3-4-5-6-7-8-9-10 ثم الولد Jack و البنت Queen و الشيخ King. و الأص قد يكون أعلى من الشيخ مرتبة أو أقل من 2 على حسب اللعبة التي تلعبها.

إن أردنا تعريف كائن جديد لتمثيل بطاقات اللعب، فمن الواضح أن خصاله ستكون: rank و suit (مرتبة و شكل)، لكن اختيار نمط الخصال ليس واضحا كالخصال نفسها. أحد الاحتمالات هو استخدام محارف بها كلمات كـ Spade للأشكال و Queen للمراتب. لهذا الاحتمال مشاكل، إحداها صعوبة مقارنة البطاقات لنرى أيها أعلى مرتبة أو شكلا.

طريقة بديلة هي استخدام الاعداد الصحيحة لترميز المراتب و الاشكال. الترميز هنا يعني أننا سنخطط بين الارقام و الاشكال، أو بين الارقام و المراتب. ليس المقصود بهذا النوع من الترميز السرية (و إلا سنسميه تشفيرا).

على سبيل المثال، يظهر الجدول التالي الاشكال و الاعداد الصحيحة المقابلة لها:

$$
\begin{aligned}
&\text{Spades} &&\mapsto 3\\
&\text{Hearts} &&\mapsto 2\\
&\text{Diamonds} &&\mapsto 1\\
&\text{Clubs} &&\mapsto 0
\end{aligned}
$$

هذا الترميز يسهل علينا مقارنة البطاقات لأن الشكل الاعلى يقابله عدد أكبر، فيمكننا مقارنة الاشكال بمقارنة ما يرمز اليها.

التخطيط للمراتب واضح بما فيه الكفاية، رقم كل بطاقة سيرمز للعدد الصحيح الذي يمثله، و للصور:

Jack $\mapsto$11

Queen $\mapsto$12

King $\mapsto$13

استخدمت الشكل$\mapsto$ لكي أقول بأن هذه التخطيطات ليست جزءا من بايثون. بل جزء من تصميم البرامج، و هي لا تظهر صريحة في النصوص البرمجية.

تعريف الفئة لـ Cards سيبدو كالتالي:

```
class Card(object):
   """Represents a standard playing card"""

   def __init__(self, suit=0, rank=2):
      self.suit = suit
      self.rank = rank
```

و كالعادة، تأخذ طريقة init برمتر إختياري لكل خصلة و الافتراضي هو 2 للسباتي

و لكي تنشئ بطاقة عليك نداء Card مع شكل و مرتبة البطاقة التي تريدها

```
queen_of_diamonds = Card(1, 12)
```

18.2 خصال الفئة

لكي نطبع كائن البطاقة بالشكل الذي يقرأه الناس بسهولة، علينا التخطيط من الترميز العددي إلى المرتبة و الشكل المقابلين له من الطبيعي أن يكون السبيل إلى ذلك قوائم من المحارف. تعين تلك القوائم إلى خصال فئة:

```
# inside class Card:

   suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
   rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

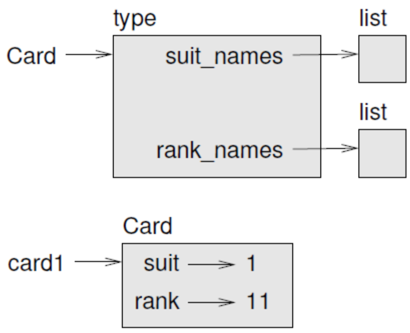
   def __str__(self):
      return '%s of %s' % (Card.rank_names[self.rank],
                           Card.suit_names[self.suit])
```

المتغيرات مثل suit_names و rank_names المعرفة داخل الفئة لكن ليس داخل أي طريقة تسمى **خصال الفئة** لأنها ترتبط بكائن الفئة Card.

المصطلح يفرق بين متغيرات مثل suit و rank و التي تسمى خصال التجليات لأنها تختص بتجلية بعينها.

كلا النوعين من الخصال يكون الوصول اليها بالتنويت مثلا __str__ فيها self كائن بطاقة، و self.rank هي مرتبة البطاقة و بنفس الطريقة فإن Card هو كائن بطاقة و Card.rank_names هي قائمة بالمحارف المتعلقة بهذه الفئة.

و لكل فئة suit و rank تخصها، لكن هناك نسخة واحدة فقط من suit_names و من rank_names



الشكل 18.1 رسم الكائن

أن وضعنا كل هذه العبارات معا `Card.rank_names[self.rank]` ستعني "استخدم الخصلة `rank` للكائن `self` كمؤشر في قائمة `rank_names` من الفئة `Cards` ثم اختر المحارف المناسبة"

العنصر الاول من `rank_names` يكون `None` لعدم وجود بطاقة مرتبتها صفر. بإدراج `None` كحارس على منزلة نحصل على توجيه له ميزة جميلة، و هي أن المؤشر 2 يوجه إلى المحارف 2 و هكذا. لتجنب فركة كهذه كان يتوجب علينا استخدام قاموس بدلا من قائمة.

بالطرائق التي لدينا حتى الان يمكننا خلق و طباعة البطاقات:

```
>>> card1 = Card(2, 11)
>>> print card1
Jack of Hearts
```

18.3 مقارنة البطاقات

هنالك مؤثرات نسبية للأنماط الجاهزة (<, > , == الخ) تقارن بين القيم و تقرر إذا ما كانت احداها أكبر أو أصغر أو تساوي الاخرى. بالنسبة لأنماط المستخدم يمكننا تخطي السلوك الاصلي لهذه المؤثرات باستخدام طريقة تدعى `__cmp__`

تأخذ `__cmp__` برمترين `self` و `other` ، ثم ترجع رقما موجبا إن كان الكائن الأول أكبر من الثاني و سالبا كان الثاني أكبر و صفرا إن تساوا.

الترتيب الصحيح للبطاقات غير واضح مثلا، أيها أفضل: 3 سباتي أم 2 ديناري؟ فواحد منها أعلى مرتبة لكن الثاني أعلى شكلا. إذن للمقارنة بين البطاقات عليك أن تحدد ما هو الاهم، المرتبة أم الشكل.

قد تعتمد الاجابة على أي لعبة تلعب، لكن لنبسط الأمور و نقول بأن الشكل هو الأهم، إذن فكل البستوني أعلى مرتبة من من كل الديناري و هكذا.

أما و قد استقرينا على هذا الترتيب، فيمكننا الان كتابة `__cmp__` :

```
# inside class Card:


  def __cmp__(self, other):
     # check the suits
     if self.suit > other.suit: return 1
     if self.suit < other.suit: return -1

     # suits are the same... check ranks
     if self.rank > other.rank: return 1
     if self.rank < other.rank: return -1
```

```
        # ranks are the same... it's a tie
        return 0
```

يمكن كتابة نفس النص لكن بإيجاز باستخدام مقارنات توبل:

```
# inside class Card:

    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

للإقتران الجاهز cmp نفس واجهة الطريقة __cmp__ : يأخذ قيمتين و يرجع رقما موجبا إن كانت الاولى أكبر و سالبا إن كانت الثانية أكبر و صفر إن تساوتا.

في بايثون3 اختفت طريقة __cmp__ و كذلك cmp ، لذا عليك استبدالها بـ __lt__ التي ترجع True إن كانت self أصغر من other بإمكانك تطبيق __lt__ باستخدام توبل و مؤثر < (أصغر)

تمرين 18.1 أكتب طريقة __cmp__ لكائنات Time. تلميح: يمكنك استخدام مقارنة توبل، لكن يمكنك الاخذ بالاعتبار استخدام طرح الأعداد الصحيحة.

18.4 الشدات

الان و قد أصبح لدينا بطاقات اللعب Cards حان الوقت لتعريف الشدات Decks. بما أن الشدة مكونة من بطاقات فمن الطبيعي أن تحتوي كل شدة على قائمة من البطاقات كخصلة.

التالي هو تعريف فئة Deck. توجد طريقة init الخصلة cards و توجد مجموعة الاثنين و خمسين بطاقة النموذجية :

```
class Deck(object):

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

أسهل طريقة لمكاثرة الشدة هي استخدام حلقة عشية. ترقم الحلقة الخارجية الاشكال من 0 إلى 3، الحلقة الداخلية ترقم المراتب من 1 إلى 13. كل تكرار يوجد بطاقة جديدة مع المرتبة و الشكل، ثم تضمها إلى self.cards

18.5 طباعة البطاقات

هاك طريقة `__str__` للشدة:

```
#inside class Deck:

   def __str__(self):
      res = []
      for card in self.cards:
         res.append(str(card))
         return '\n'.join(res)
```

تشرح هذه الطريقة اسلوب فعال لمراكمة المحارف الكبيرة: بناء قائمة من المحارف ثم استخدام `join`. الاقتران الجاهز `str` ينادي الطريقة `__str__` على كل بطاقة و يرجع محارف تمثلها.

بما أننا استدعينا `join` على محارف سطر جديد، فستُفصل البطاقات بسطر جديد. هكذا ستبدو النتيجة:

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

حتى و إن بدت النتيجة كـ 52 سطرا، فهي لا تزال محارف طويلة واحدة تحتوي على سطور جديدة.

18.6 أضف، إحذف، إخلط و رتب

لابد، للتعامل مع بطاقات اللعب، من طريقة تمكنك من سحب بطاقات من الشدة و إعادتها، تمنحنا الطريقة `pop` طريقة مقبولة لهذه المهمة:

```
#inside class Deck:

   def pop_card(self):
      return self.cards.pop()
```

بما أن `pop` تسحب اخر بطاقة في القائمة، فإننا نتعامل مع الشدة من الأسفل. يعتبر سحب البطاقات من الاسفل غشا (فت من تحت)، لكن في هذا السياق فهو جائز.

لإضافة بطاقات سنستفيد من `append`:

```
#inside class Deck:

   def add_card(self, card):
      self.cards.append(card)
```

هكذا اقترانات ، تستخدم اقترانات اخرى دون القيام بأي عمل بنفسها تسمى في الإنجليزية قشرة، و أصلها من النجارة حيث تكسى الطبقة الخارجية من رخيص الخشب بأخرى من ثمينه. (في نابلس قشرة: نحاس مطلي بالذهب).

نحن هنا نعرِّف طريقة واهنة للتعبير عن عملية على قائمة من التعبيرات التي تناسب الشدات.

كمثال اخر، يمكننا كتابة طريقة شدة اسمها `shuffle` تستخدم الاقتران `shuffle` من مديول `random` :

```
# inside class Deck:
   def shuffle(self):
      random.shuffle(self.cards)
```

لا تنس استيراد random .

تمرين 18.2 اكتب طريقة شدة اسمها sort تستخدم الطريقة sort لترتيب البطاقات في deck.sort باستخدام طريقة __cmp__ التي عرفناها لتحديد كيفية الترتيب.

18.7 التوريث

أكثر الخصائص اللغوية تعلقا بالبرمجة كائنية المنحى هي التوريث. التوريث هي القابلية لتعريف فئة جديدة تكون نسخة معدلة من فئة موجودة أصلا.

سميت "توريثا" لأن الفئة الجديدة ترث طرائق اللغة الموروثة. و لمد هذا التشبيه إلى أقصاه، فالمورِّث يسمى اب و الوارث يسمى ابن.

لتوضيح هذا المفهوم، لنقل أنك تريد فئة تسمى فئة hand . و هي البطاقات التي يحملها اللاعب في يده. الفئة مثلها مثل الشدة، كلاهما يقوم على مجموعة من البطاقات، و كلاهما بحاجة إلى عمليات، كإضافة و حذف البطاقات .

لكن الفئة تختلف عن الشدة أيضا، فهناك عمليات نريد أن نقوم بها على الفئة لكنها بلا معنى للشدة، مثلا يمكننا في البوكر مقارنة فئتين لنر أيهما سترتبح. في البردج يمكننا حساب احتمالات الربح لفئة ما لأجل المراهنة.

وصف العلاقة بين الفئات (متشابهة لكن باختلاف) ينطبق على التوريث.

تعريف الفئة الابن كتعريف أي فئة أخرى، عدا عن أن الفئة الأب تكون بين قوسين:

```
class Hand(Deck):
   """Represents a hand of playing cards"""
```

يشير هذا التعريف إلى أن hand ترث من deck، مما يعني أنه بإمكاننا استخدام طرائق مثل pop_card و Hands و كذلك Decks.

رغم أن Hand ترث أيضا __init__من Deck ، إلّا أنها في الحقيقة لا تفعل ما نريدها أن تفعل: فبدلا من إكثار الفئة بالـ 52 بطاقة جديدة، على طريقة init للفئات تهيئة cards بقائمة فارغة.

إن زودنا الفئة Hand بطريقة __init__ ستتخطى تلك التي في فئة Deck :

```
# inside class Hand:

   def __init__(self, label=''):
      self.cards = []
      self.label = label
```

إذن فعندما تنشئ فئة، سيستدعي بايثون طريقة init:

```
>>> hand = Hand('new hand')
>>> print hand.cards
[]
>>> print handlabel
```

```
new hand
```

إلا أن طرق أخرى قد وُرِثت عن Deck ، إذن فيمكننا استخدام pop_card و add_card للفت:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print hand
King of Spades
```

من الطبيعي أن تكون الخطوة التالية هي كبسلة هذا النص في طريقة، و لتُسمّى move_cards:

```
#inside class Deck:

   def move_cards(self, hand, num):
      for i in range(num):
         hand.add_card(self.pop_card())
```

تأخذ move_cards قرينتين: كائن Hand و عدد البطاقات للفت، و تعدل على self و hand و ترجع None.

في بعض الالعاب تنتقل البطاقات من يد لاعب إلى يد اخر، و أحيانا من يد اللاعب إلى الشدة. يمكنك استخدام move_cards لأي من هذه الحالات: فيمكن لـ self أن تكون Deck و أن تكون Hand، و يمكن لـ hand (بغض النظر أن الاسم يعني فتة) أن تكون شدة Deck.

تمرين 18.3 أكتب طريقة Deck (شدة) و سمها deal_hands تأخذ برمترين، عدد الفتات و عدد البطاقات في كل فتة، ثم تنشئ كائن Hand جديد، ثم تفت عدد الكروت المناسب في كل فتة، ثم ترجع قائمة بكائنات Hand.

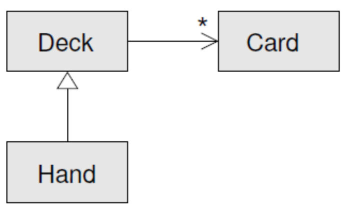
التوريث خاصية مفيدة. فنصوص بعض البرامج التي تُكرِّر العمل تصبح أكثر أناقة مع التوريث. التوريث يسهِّل إعادة استخدام النصوص البرمجية، بسبب إمكانية تهذيب سلوك الفئات الآباء لتفي بمتطلباتك دون الحاجة إلى تعديلها. و في بعض الاحيان يعكس بناء التوريث البناء الطبيعي للمشكلة، مما يجعل البرنامج أسهل للفهم.

الا أنه من ناحية ثانية، فالتوريث قد يجعل قراءة البرامج أصعب. فعند استدعاء طريقة، يكون من غير الواضح أحيانا أين يمكن العثور على تعريفها، فالنص المتعلق بها قد يكون متفرقا في عدة مديولات. ثم إن الكثير من الاشياء التي يمكن عملها بالتوريث يمكن عملها بدونه، و أحيانا تكون أفضل بدونه.

18.8 رسم الفئة

لقد رأينا حتى الان الرسم المستف الذي يبين حالة البرنامج، و رسم الكائن الذي يبين خصال الكائن و قيمها. تمثل تلك الرسومات لمحة لحظية من تنفيذ البرنامج، لذلك فهي تتغير دائما خلال عمل البرنامج.

تلك الرسومة مفصلة جدا، و لأهداف معينة يبالغ في التفصيل رسم الفئة، في المقابل، يكون تمثيلا مختصرا لبناء البرنامج فبدلا من إظهار الكائنات المنفردة، يظهر الفئات و العلاقة بينها.



الشكل 18.2: رسم الفئة

هنالك عدة أنواع من العلاقات بين الفئات:

- كائنات داخل فئة بها مرجعيات لكائنات في فئة أخرى مثلا، كل Rectangle يحتوي على مرجع لـ Point، و كل Deck يحتوي على مراجع لعدة Card هذه العلاقات تسمى HAS-A أي (a Rectangle has a Point).
- قد ترث إحدى الفئات من أخرى و هذه العلاقة تسمى IS-A أي (a Hand is a kind of Deck).
- قد تعتمد إحدى الفئات على الاخرى من ناحية أن التغيير على واحدة يتطلب تغييرا على الفئة الاخرى.

رسم الفئة هو تمثيل تصويري لتلك العلاقات مثلا، الشكل 18.2 يظهر العلاقة بين Card و Deck و Hand.

السهم ذو الرأس المثلث المفرغ يمثل علاقةIS-A ، و في هذه الحالة يعني أن Hand ترث من Deck.

السهم العادي يمثل علاقة HAS-A ، و في هذه الحالة فإن لدى Deck مرجاع لكائنات Card.

النجمة * قرب السهم للتكرار، فهي تقول لنا عدد البطاقات في الشدة، و قد يكون التكرار رقم عادي مثل 52، أو نطاقا كـ 5...7، أو قد تكون نجمة * و التي تعني بأنه يمكن أن يكون للشدة أي عدد من البطاقات.

الرسوم المفصلة أكثر قد تحتوي على قائمة بالبطاقات، لكن في العادة لا تضاف الانماط الجاهزة في رسوم الفئات.

تمرين 18.4 اقرأ `TurtleWorld.py` و `World.py` و `Gui.py` ثم أرسم رسم فئة يوضح العلاقة بينها

18.9 علاج الأخطاء

قد يجعل التوريث من علاج الأخطاء عصر مخ حقيقي. لأنك عندما تستدعي طريقة على كائن فأنت لا تعلم بالضبط أي طريقة ستُستَدعى.

إفرض أنك تكتب اقترانا يعمل مع كائنات `Hand`. ستريد العمل مع كافة أنواع الفئات، كفئة البوكر و البردج...الخ. إن استدعيت طريقة كـ `shuffle` فقد تحصل على تلك الطريقة المعرفة في `Deck` ، لكن أن تخطت أي من الفئات الفرعية هذه الطريقة فستحصل على نسخة الطريقة من تلك الفئة.

في كل مرة لاتكون متأكدا من سريان تنفيذ برنامجك، يكون الحل الابسط هو إضافة عبارة `print` في بداية الطريقة المقصودة. فإن طبَعَت `Deck.shuffle` الرسالة التي تقول شيئا مثل `Running Deck.shuffle` فسيمكنك متابعة سريان البرنامج.

أو يمكنك استخدام هذا الاقتران، و هو يأخذ كائنا و طريقة و يرجع الفئة التي زودت تعريف الطريقة:

```
def find_defining_class(obj, meth_name):
   for ty in type(obj).mro():
      if meth_name in ty__dict__:
         return ty
```

و هذا مثال:

```
>>> hand = Hand()
>>> print find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

اذن فطريقة `shuffle` لهذه الـ `Hand` أتت من `Deck`.

`find_defining_class` تستخدم طريقة `mro` للحصول على قائمة بكائنات الفئة (أنماط الفئة) حيث سيبحث

فيها عن الطرائق. MRO اختصار method resolution order.

إليك اقتراحا حول تصميم البرنامج: عندما تتخطى طريقة ما فيجب أن تكون واجهة الطريقة الجديدة هي نفسها واجهة القديمة. و عليها أخذ نفس البرمترات، و إرجاع نفس الانماط، و اتباع نفس الشروط المسبقة. إن اتبعت هذا المبدأ فسترى بأن كل اقتران صمم ليعمل مع تجلية لفئة مثل Deck، سيعمل مع تجليات الفئات الفرعية مثل Hand أو PokerHand.

و إن خالفت هذا المبدأ فسينهار النص البرمجي مثل (اسف على التعبير) بيت من بطاقات الشدة.

18.10 كبسلة البيانات

في الفصل 16 شرحنا خطة تطوير يمكن تسميتها ب"التصميم كائني المنحى" فقد حددنا الكائنات التي نحتاجها Time, Point, Rectangle، و عرفنا الفئات التي تمثلها. كان هناك دائما ما يربط بين الكائنات و بين شيء على أرض الواقع (أو الواقع الرياضي).

لكن في في بعض الاحيان لا يكون الربط بينها و بين الواقع ظاهرا، و لا يكون واضحا أي كائنات تريد، و لا كيفية تعاملها مع بعضها. بنفس الطريقة التي استكتشفنا فيها واجهات الاقترانات عن طريق الكبسلة و التعميم، سنكتشف واجهات الفئات عن طريق كبسلة الفئات.

تحليل ماركوف من القسم 13.8 مثالا ممتازا. إن كنت قدحملت نصي من

.http://thinkpython.com/code/markov.py

سترى بأنه يستخدم متغيرين عموميين suffix_map و prefix تقرأها و تكتبها عدة اقترانات.

```
suffix_map = {}
prefix = ()
```

و لكون هذه المتغيرات عمومية فلن نتمكن من إجراء أكثر من تحليل واحد كل مرة. فعندما نقرأ نصين ستضاف البادءات و اللواحق فيهما إلى نفس هيكل البيانات (و الذي يجد ذاته مصدرا للنصوص المسلية).

لكي نجري أكثر من تحليل و نبقيها منفصلة، يمكننا كبسلة الحالة لكل تحليل في كائن. هذا ما سيبدو عليه:

```
class Markov(object):
      def __init__(self):
            self.suffix_map = {}
            self.prefix = ()
```

الخطوة التالية هي تحويل الاقترانات إلى طرائق مثلا هذه process_word:

```
      def process_word(self, word, order=2):
            if len(self.prefix) < order:
                  self.prefix += (word,)
                  return

      try:
            self.suffix_map[self.prefix]append(word)
      except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

      self.prefix = shift(self.prefix, word)
```

تحويل البرنامج بهذا الاسلوب - تغيير التصميم بدون تغير الاقتران – هو مثال اخر على التفتيت و البناء (انظر القسم 4.7).

هذا المثال أوحى لنا بخطة تطوير لتصميم الكائنات و الطرائق:

1. ابدأ بكتابة الاقترانات التي تقرأ و تكتب المتغيرات العمومية (عند الضرورة).
2. عندما تتمكن من جعل البرنامج يعمل، ابحث عن ترابطات بين المتغيرات العمومية و الاقترانات التي تستخدمها.
3. كبسل المتغيرات المتعلقة بكائن كخصال لهذا الكائن.
4. حول الاقترانات المترابطة إلى طرائق في الفئة الجديدة.

تمرين 18.5 حمل نصي من الفصل 13.8 `http://thinkpython.com/code/markov.Py`.

ثم اتبع التعليمات الآنف ذكرها لكبسلة المتغيرات العمومية كخصال للفئة الجديدة، و التي سيكون اسمها Markov. الحل: `http://thinkpython.com/code/Markov.py`.

18.11 المعاني

ترميز (رمز) encode: أن تمثل مجموعة من القيم باستخدام مجموعة قيم أخرى عن طريق الوصل بينها.

خصلة فئة Class attribute: خصلة ترتبط بكائن فئة، تعرَّف خصال الفئات داخل تعريف الفئة، لكن خارج أية طريقة.

خصلة تجلية instance attribute: خصلة ترتبط لتجلية لفئة ما.

قشورية veneer: طريقة أو اقتران توفر واجهات مختلفة لاقترانات اخرى، من دون أن يكون لها عمل حوسبي يخصها.

التوريث inheritance: القدرة على تعريف فئة جديدة تكون نسخة معدلة عن فئة معرفة سابقا.

الفئة الاب parent class: الفئة التي ترث منها الفئة الابن.

الفئة الابن child: فئة جديدة انشئت بالوراثة من فئة قائمة، تسمى أيضا فئة فرعية.

علاقة IS-A: العلاقة بين فئة الابناء و فئة الاباء.

علاقة HAS-A: علاقة بين فئتين، تحتوي تجليات إحداها مراجع لتجليات ا لفئة الاخرى.

رسم الفئة class diagram: رسم يبين الفئات في برنامج و العلاقات بينها.

تعدد الاشكال multiplicity: تنويت في رسم الفئة يظهر ، لعلاقة HAS-A ، عدد المراجع إلى تجليات من فئة أخرى.

18.12 تمارين

تمرين 18.6 التالي هو عدد الفئات المتاحة في لعبة بوكر، بترتيب تصاعدي حسب القيمة (و تنازليا حسب الاحتمال):

زوجان pair: بطاقتان لها نفس المرتبة.

أربعة أزواج two pairs: أربعة بطاقات لها نفس المرتبة.

ثلاثة من نوعها three of a kind: ثلاثة بطاقات لها نفس المرتبة.

مباشر strait: خمس بطاقات متسلسلة المرتبة (الاص يمكن اعتباره أعلى أو أدنى، أص، 2-3-4-5 هي straight، و كذلك 10-ولد-بنت-شيخ-أص، لكن لا تعتبر بنت-شيخ-أص-2-3 straight).

فلش flush: خمس بطاقات من نفس الشكل.

فُلهاوس full house: ثلاثة بطاقات من نفس المرتبة و بطاقتان من مرتبة واحدة أخرى.

فلش مباشر straight flush: خمس بطاقات متسلسلة لكن من نفس الشكل.

هدف هذه التمارين هو توقع سحب هذا البطاقات في الفئات المختلفة.

1. حمل الملفات التالية من `http://thinkpython.com/code`:
   `Cards.py`: نسخة كاملة من فئات Card و Deck و Hand التي مرت علينا في هذا الفصل.
   `PokerHand.py`: تطبيق غير مكتمل من فئة تمثل فئة بوكر و معها نص برمجي لاختبارها.
2. عندما تشغل `PokerHand` فسيقوم بفتات بوكر من 7 بطاقات و يفحص إن كان أي منها يحتوي على فلش اقرأ هذا النص البرمجي بعناية قبل المضي قدما.
3. أضف طرائق لـ `PokerHand.py` و سمها `has_pair` و `has_twopair` و هكذا، ترجع `True` أو `False` حسب موافقة الفئة مع القوانين المقابلة على النص أن يعمل مع أي الفتات التي بها أي عدد من البطاقات (رغم أن 5 و 7 هي الفتات الاكثر شيوعا في البوكر) .
4. أكتب طريقة اسمها `classify` تكتشف أعلى قيمة في التصنيف و تعدل خصلة `label` بناءا عليها مثلا فتة 7 بطاقات قد يكون بها فلش و زوجين، إذن يجب أن تسمى فلش.
5. عندما ترتاح إلى أن طرائق التصنيف تعمل، الخطوة التالية هي توقع احتمالات الفتات المختلفة اكتب اقترانا في `PokerHand.py` يخلط بطاقات شدة، و يوزعها على فتات ثم يحصي عدد مرات ظهور التصنيفات.
6. اطبع جدولا بالتصانيف و احتمالاتها شغل البرنامج عدة مرات تزيد فيها عدد الفتات إلى أن تصبح النتيجة بدقة معقولة قارن نتائجك بالقيم من `http://en.wikipedia.org/wiki/Hand_rankings`.

الحل: `http://thinkpython.com/code/PokerHandSoln.py`.

تمرين 18.7 هذا التمرين يستخدم TurtleWorld من الفصل 4، ستكتب برنامجا يمكن السلحفاة من لَعِب اللمسة، إن لم تكن تعرفها فاقرأ عن Tag من `http://enwikipediaorg/wiki/Tag_game`.

1. حمل `http://thinkpython.com/code/Wobbler.py`. و شغله سترى عالم السلحفاة و به ثلاثة سلاحف إن ضغطت زر RUN ستمشى السلاحف عشوائيا.

2. اقرأ النص البرمجي و تأكد من استيعابك لطريقة عمله. ترث فئة `Wobbler` من `Turtle` ، مما يعني أن طرائق `Turtle` و هي `(lt, rt, fd, bk)` ستعمل على `Wobbler`.
   تُستدعى طريقة `step` من قبل `TurtleWorld`و بدورها تَستدعِي `steer` التي توجه السلحفاة إلى الاتجاه المطلوب، و `wobble` التي تقوم بالتفافة عشوائية بالتناسب مع بلادة السلحفاة، و `move` و التي تحرك السلحفاة إلى الامام، بضعة بكسلات حسب سرعتها.
3. اخلق ملفا اسمه `Taggerpy` استورد كل شيء من `Wobbler` ثم عرف فئة اسمها `Tagger` ترث من `Wobbler`نادي `make_wrld` ممرا كائة الفئة `Tagger` كقرينة.
4. اضف طريقة `steer` إلى `Tagger` لكي تتخطى تلك التي في `Wobbler`. كنقطة بداية اكتب نسخة توجه السلحفاة إلى نقطة الاصل دائما. تلميح: استخدم الاقتران الرياضي `atan2` و خصال السلحفاة `x, y` و `heading`.
5. عدل على `steer` بحيث تظل السلحفاة ضمن الحدود. و لعلاج الأخطاء استفد من زر `Step` و الذي يستدعي `step` مرة واحدة على كل سلحفاة.

6. عدل على steer بحيث توجه كل سلحفاة وجهها نحو السلحفاة الاقرب اليها. تلميح: للسلاحف خصلة اسمها world و هي مرجع لـ TurtleWorld الذي تعيش فيه، و لـ TurtleWorld خصلة اسمها animals و هي قائمة بكل السلاحف في ذلك الكون.
7. عدل على steer بحيث تلعب السلاحف اللمسة (Tag). لك أن تضيف طرائق لـ Tagger و يمكنك تخطي steer و __init__ ، لكن لا يمكنك التعديل على أو تخطي wobble ,step أو move . و أيضا يسمح لـ steer بتغيير اتجاه السلحفاة لكن ليس موقعها.

عدل القوانين و طريقة steer لتجويد اللعبة مثلا، يجب تمكين السلحفاة البطيئة من لمس السلحفاة السريعة في اخر المطاف.

الحل: http://thinkpython.com/code/Tagger.py.

الفصل التاسع عشر

دراسة حالة: تيكنتر Tkinter

19.1 واجهة المستخدم الرسومية GUI

معظم البرامج التي رأينا لحد الان نصية الطابع. لكن الكثير من البرامج تستخدم و اجهات مستخدم رسومية، و تسمى أيضا GUI.

يوفر لنا بايثون عدة خيارات لكتابة برامج GUI ، من ضمنها wxPython و Tkinter و QT. لكل منها حسناته و نقاط ضعفه و لهذا السبب لم يستقر بايثون على أحدها ليكون الخيار النموذجي.

الخيار الذي سأقدمه هنا هو Tkinter لأنني أعتقد بأنه الأسهل للمبتدئ. كل المفاهيم في هذا الفصل تنطبق على مديولات الـ GUI الاخرى.

هنالك الكثير من الكتب و صفحات الويب تتعلق بتكنتر، أحد أفضلها هو "مقدمة لتكنتر" لفردريك لَنض.

لقد كتبت مديولا اسمه `Gui.py` يأتي مع سومبي. يوفر واجهة مبسطة إلى الاقترانات و الفئات في تكنتر و الامثلة في هذا الفصل مبنية على هذا المديول.

هنا مثال بسيط يخلق و يظهر واجهة مستخدم رسومية:

لإيجاد واجهة مستخدم رسومية عليك استيراد `Gui` من سوَمبي:

```
from swampy.Gui import *
```

أو قد يبدو كهذا (حسب كيف نصبت سومبي):

```
from Gui import *
```

ثم عليك عمل تجلية من كائن Gui:

```
g = Gui()
g.title('Gui')
g.mainloop()
```

عندما تشغل هذا النص سترى نافذة رمادية بعنوان `Gui`. تشغل حلقة `mainloop` حلقة انتظار الحدث(event loop)، و التي تنتظر من المستخدم القيام بشيء ما لترد عليه. و هي حلقة لا منتهية، تظل تعمل إلى أن يغلق المستخدم النافذة أو يضغط Control – C، أو يفعل شيئا يجعل البرنامج يتوقف.

لا تقوم هذه الواجهة بأي شيء لأنها لا تملك أية وشائط Widgets. الوشائط هي العناصر التي تكون واجهة المستخدم الرسومية و من ضمنها:

- **الزر Button**: وشيطة تحتوي على كتابة أو صورة و تقوم بعمل ما عندما تضغط.
- **القماشة Canvas**: مساحة تعرض خطوطا أو مربعات أو دوائر أو أي شكل اخر.

- **المدخِلة Entry**: مساحة حيث يطبع المستخدم الكتابات.
- **شريط تمرير Scrollbar**: وشيطة تحدد الجزء المرئي من وشيطة أخرى.
- **إطار Frame**: حاوية، في الغالب مرئية، تحتوي على وشائط أخرى.

المستطيل الرمادي الذي رأيته عندما أوجدت واجهة المستخدم كان إطارا. و كلما أنشأت وشيطة ستضاف الى هذا الإطار.

19.2 الازرار و معاودات النداء Buttons and callbacks

توجِد الطريقة bu وشيطة زر:

```
button = g.bu(text='Press me')
```

القيمة المرتجعة من bu هي كائن Button. و الزر الذي يظهر في الاطار هو تمثيل رسومي لهذا الكائن، و يمكنك التحكم بالزر عن طريق استدعاء طرائق عليه.

تأخذ bu حوالي 32 برمتر للتحكم بمظهر و وظائف الزر. و هذه البرمترات تسمى هنا خيارات options. و بدلا من تزويد القيم لكل الـ 32 خيارا. بوسعك استخدام قرائن كلمات مفاتيح مثل text = 'Press me' لتحديد الخيارات التي تريد و تترك الباقي للقيم الافتراضية.

عند اضافتك وشيطة إلى الاطار فإنه يتقلص لها " shrink-wrapped". أي أن الاطار يتقلص إلى حجم الزر. و إن أضفت وشائط اخرى سيكبر حجم الاطار بمقدار يكفي لاستضافتها.

الطريقة la توجد وشيطة ملصق Label :

```
label = g.la(text='Press the button')
```

بالوضع الأولي يكدس تكنتر الوشائط من الاعلى إلى الاسفل، و يوسطها. سنرى قريبا كيف نتخطى هذا السلوك.

إن ضغطت الزر فسترى بأنه لا يقوم بالكثير. و ذلك لأنك "لم توصل أسلاكه بعد" أي لم تقل له ما يفعل.

الخيار الذي يتحكم بالزر هو command . و قيمة command هو اقتران سيتم تنفيذه عندما يُضغط الزر. هنا مثال ينشئ ملصقا جديدا:

```
def make_label():
   g.la(text='Thank you')
```

الان يمكننا انشاء زر له هذا الخيار:

```
button2 = g.bu(text='No, press me!', command=make_label)
```

فعندما تضغط هذا الزر عليه أن ينفذ اقتران make_label و يجب أن يظهر ملصقا جديدا.

قيمة الخيار command هي كائن اقتران، و هو ما يعرف"بمعاود النداء : callback"، لأنك عندما تنادي bu لإنشاء الزر، فإن سريان التنفيذ سيعاود النداء عندما يضغط المستخدم الزر.

هذا النوع من السريان هو من صفات البرمجة المقادة بالاحداث event-driven programming. أفعال المستخدم مثل ضغط زر و الضربات على لوحة المفاتيح تسمى أحداثا، في البرمجة المقادة بالاحداث يعتمد سريان البرنامج على أفعال المستخدم أكثر من المبرمج.

التحدي في اسلوب البرمجة هذا هو بناء مجموعة من الوشائط و بردود نداء تعمل بشكل صحيح (أو تصدر رسالة خطأ واضحة و مناسبة) لأي تسلسل من الافعال التي يقم بها المستخدم.

تمرين 19.1 اكتب برنامجا ينشئ واجهة مستخدم بها زر. عند ضغطه، عليه أن ينشئ زرا اخر. و عند ضغط الاخر ينشئ ملصقا يقول "Nice job!".

ما الذي سيحدث إن ضغط الزر أكثر من مرة؟

الحل: http://thinkpython.com/code/button_demo.py

19.3 وشائط قماشة الرسم

قماشة الرسم هي من أكثر الوشائط تنوعا، و هي تنشئ مساحة لرسم الخطوط و الدوائر و الاشكال الاخرى. إن حللت التمرين 15.4 فأنت على دراية بقماشة الرسم.

الطريقة ca تنشئ Canvas جديدة:

```
canvas = g.ca(width=500, height=500)
```

width و height. هي أبعاد قماشة الرسم بالبكسل.

لا يزال بوسعك بعد إنشاء الوشيطة تغيير قيم الخيارات بالطريقة config. مثلا خيار bg سغير لون الخلفية:

```
Canvas.config(bg='white')
```

قيمة bg هي محارف باسم اللون. مجموعة الالوان المتاحة تختلف باختلاف تطبيقات بايثون، لكن كل التطبيقات توفر على الاقل:

```
white    black
red      green    blue
cyan     yellow   magenta
```

الاشكال المرسومة على قماشة الرسم تسمى عناصر، items ، فمثلا طريقة Canvas المسماة circle ترسم، للغرابة،دائرة:

```
item = canvas.circle([0,0], 100, fill='red')
```

القرينة الاولى هي زوجين من الاحداثيات تحدد مركز الدائرة، و الثانية هي نصف القطر.

يوفر Gui.py نظام المستوى الديكارتي النموذجي. فيه نقطة الاصل هي وسط قماشة الرسم، و المحور الصادي الموجب يتجه إلى الاعلى. و هو على خلاف الانظمة الاخرى حيث نقطة الاصل تكون في الزاوية اليسرى العليا و المحور الصادي الموجب يتجه إلى الاسفل.

الخيار fill يعني أن الدائرة ستملأ باللون الاحمر.

و القيمة المرتجعة من circle هي كائن عنصر يوفر طرائق للتعديل على العنصر على القماشة. مثلا يمكنك استخدام config لتغيير أي من خيارات الدائرة:

```
item.config(fill='yellow', outline='orange', width=10)
```

سمك خط محيط الدائرة هو width و الوحدة هي بكسل و لونه هو outline

تمرين 19.2 اكتب برنامجا ينشئ قماشة و زر. عندما يضغط المستخدم الزر، يجب أن يرسم دائرة على القماشة.

19.4 تسلسلات الاحداثيات

طريقة rectangle تأخذ تسلسلا من الاحداثيات لتحدد الزوايتين المتقابلتين في المستطيل. يرسم المثال التالي

مستطيل ركنه السفلى الايسر في نقطة الاصل و الركن المقابل في النقطة (200,100) :

```
canvas.rectangle([[0, 0], [200, 100]],
                 fill='blue', outline='orange', width=10)
```

تسمى هذا الطريقة (تحديد الركنين) بالصندوق المحيط bounding box، لأن النقطتين تحوطان المستطيل.

تأخذ `ovel` صندوقا محيطا و ترسم بيضاويا ضمن حدود المستطيل:

```
canvas.oval([[0, 0], [200, 100]], outline='orange', width=10)
```

و تأخذ `line` تسلسلا من الاحداثيات لترسم خطا يصل بين النقاط يرسم المثال التالي ضلعي مثلث:

```
canvas.line([[0, 100], [100, 200], [200, 100]], width=10)
```

و `polygon` تأخذ نفس القرائن إلا أنها تخط اخر ضلع في المضلع (ان تتطلب الامر) و تملأه:

```
Canvas.polygon([[0, 100], [100, 200], [200, 100]],
fill='red', outline='orange', width=10)
```

19.5 المزيد من الوشائط

يوفر تكنتر وشيطتين تسمحان للمستخدم بطباعة النصوص: المدخِلة Entry و هي لسطر واحد، و وشيطة Text و هي للسطور المتعددة.

تنشئ `en` مدخلة جديدة:

```
entry = g.en(text='Default text')
```

خيار `text` يسمح لك بإدخال النص في المدخلة عند إنشائها، أما طريقة `get` فترجع محتوى المدخلة (و الذي قد يكون غيره المستخدم):

```
>>> entry.get()
'Default text'
```

تنشئ ننشئ `te` وشيطة Text:

```
text = g.te(width=100, height=5)
```

هنا `width` و `height` هما أبعاد الوشيطة بالحروف و السطور.

تضع `insert` نصا في وشيطة Text :

```
text.insert(END, 'A line of text')
```

و `END` هي مؤشر مخصوص، يؤشر إلى آخر حرف في وشيطة Text .

يمكنك تحديد حرف باستخدام المؤشر المنقوط مثل `1.1`، يكون رقم السطر قبل النقطة و رقم العمود بعدها. في المثال التالي تضاف الحروف `nother` الى ما بعد الحرف الاول في السطر الاول:

```
>>> text.insert(1.1, 'nother')
```

تقرأ طريقة `get` النص من الوشيطة و تأخذ مؤشري البداية والنهاية كقرائن.

في المثال التالي يُرجَع كل النص من الوشيطة، و من ضمنه حرف السطر الجديد:

```
>>> text.get(0.0, END)
'Another line of text\n'
```

تحذف طريقة `delete` النص من الوشيطة. هذا المثال يحذف كل الحروف ما عدا الاول و الثاني:

```
>>> text.delete(1.2, END)
>>> text.get(0.0, END)
```

'An\n'

تمرين 19.3 عدل حلك للتمرين 19.2 بإضافة مدخلة و زر ثان. عندما يضغط المستخدم الزر الثاني يجب أن يقرأ اسم اللون من المدخلة. و يستخدمه لتغيير لون الدائرة. استخدم `config` لتعديل الدائرة الموجودة، لا تنشئ دائرة جديدة.

على برنامجك أن يكون قادرا على التعامل مع الحالات التي يحاول فيها المستخدم تعديل لون دائرة لم تنشأ، و مع الحالات التي يكون فيها اللون غير صالح.

19.6 تغليف الوشائط

حتى الان لا زلنا نكدس الوشائط في عمود واحد، لكن في معظم واجهات المستخدم الرسومية يكون النسق معقدا أكثر من هذا. مثلا الشكل 19.1 يظهر نسخة بسيطة من عالم السلحفاة (أنظر الفصل 4)

يقدم لك هذا القسم النص الذي ينشئ هذه الواجهة، و هو مقسم إلى سلسلة من الخطوات. يمكنك تحميل المثال كاملا من http://thinkpython.com/code/SimpleTurtleWorld.py.

في أعلى مستوى فيها، تحتوي هذه الواجهة على وشيطتين – قماشة و إطار – مرتبتان في صف واحد. إذن فالخطوة الاولى هي إنشاء صف.

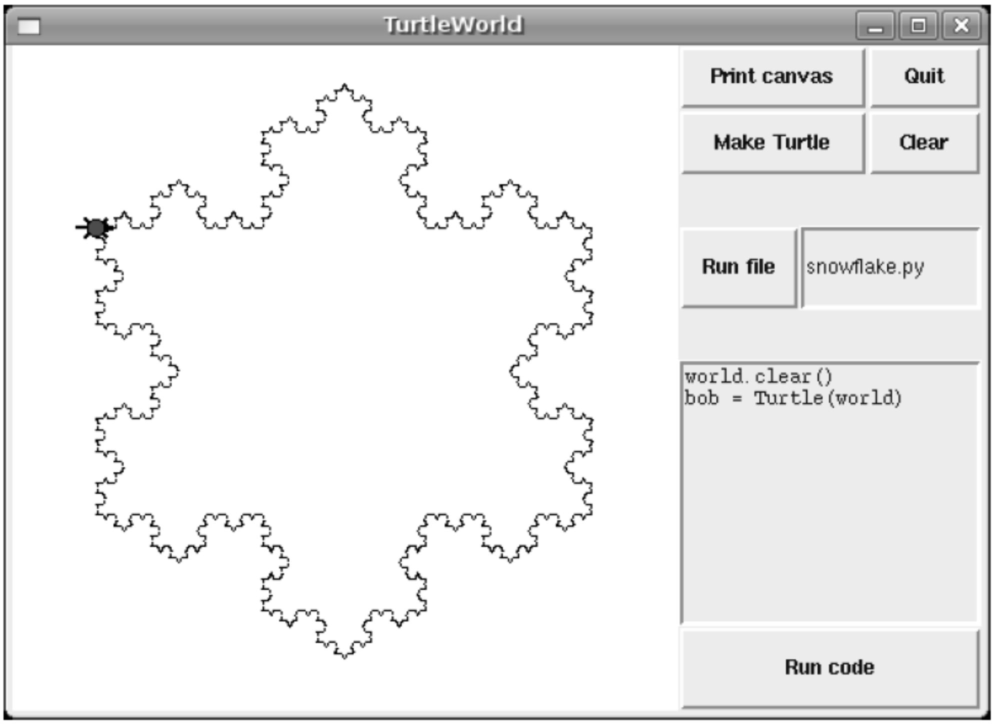
```
class SimpleTurtleWorld(TurtleWorld):
   """This class is identical to TurtleWorld, but the code that
   lays out the GUI is simplified for explanatory purposes"""

   def setup(self):
      self.row()
      ...
```

الاقتران الذي ينشئ و يرتب الوشائط هو `setup`. عملية ترتيب الوشائط في واجهة المستخدم تسمى تغليف packing.

تنشئ `row` إطار صف و تجعله "الاطار الحالي". و إلى أن يغلق هذا الاطار أو يفتح إطار جديدا، فكل الوشائط ستكدس في صف.

و هنا النص الذي ينشئ القماشة و إطار العمود الذي سيحتوي الوشائط الاخرى:



الشكل 191: عالم السلحفاة بعد تشغيل نص ندفات الثلج

```
self.canvas = self.ca(width=400, height=400, bg='white')
self.col()
```

الوشيطة الاولى في العمود هي إطار شبكي، و تحتوي على أربعة أزرار مرتبة إثنان و إثنان:

```
self.gr(cols=2)
self.bu(text='Print canvas', command=self.canvas.dump)
self.bu(text='Quit', command=self.quit)
self.bu(text='Make Turtle', command=sel.fmake_turtle)
self.bu(text='Clear', command=self.clear)
self.endgr()
```

تنشئ `gr` الشبكة، و قرينتها هي عدد الاعمدة. ترتَّب الوشائط فيها من اليسار إلى اليمين و من الاعلى إلى الاسفل.

يستخدم الزر الاول `self.canvas.dump` لمعاودة النداء، و الزر الثاني للإنتهاء. و هذه الطرائق محيطة، pound methods، مما يعني أنها مرتبطة بكائن محدد. و عند استدعائها تُستدعى على الكائن.

الوشيطة التالية في العمود هي إطار صف يحتوي على زر و مدخلة:

```
self.row([0,1], pady=30)
self.bu(text='Run file', command=self.run_file)
self.en_file = self.en(text='snowflake.py', width=5)
self.endrow()
```

القرينة الاولى لـ `row` هي قائمة تحدد كيف تنظِّم المساحات الزائدة بين الوشائط. القائمة `[0,1]` تعني أن كل المساحة الزائدة تعطى للوشيطة الثانية، و هي المدخلة. إن شغلت هذا النص و عدلت حجم النافذة سترى بأن حجم المدخلة سيتغير أما حجم الزر فلا.

الخيار `pady` يجعل هذا الصف يتكئ على محوره الصادي، مضيفا له 30 بكسل من المساحة من الاعلى و مثلها من الاسفل.

endrow تنهي هذا الصف من الوشائط، فالوشائط التالية ستتغلف في إطار العمود

تحتفظ Gui.py بتكديس من الاطارات:

- عندما تستخدم row، col ، أو gr لإنشاء إطار، فإنه يذهب إلى اعلى التكديس و يصبح الاطار الحالي.
- عندما تستخدم endrow, endcol أو endgr لإنشاء إطار فسيبرز خارج التكديس، و الاطار السابق يصبح الاطار الحالي.

تقرأ الطريقة read_file محتويات المدخلة، تستخدمها كاسم ملف و تقرأ محتوياته ثم تمررها إلى run_code.self.inter الذي هو كائن مفسِّر، يعرف كيف يأخذ المحارف و ينفذها كأنها نص برمجي من بايثون.

```
def run_file(self):
    filename = self.en_file.get()
    fp = open(filename)
    source = fp.read()
    self.inter.run_code(source, filename)
```

آخر وشيطتين كانتا وشيطة نص و وشيطة زر.

```
    self.te_code = self.te(width=25, height=10)
    self.te_code.insert(END, 'world.clear()\n')
    self.te_code.insert(END, 'bob = Turtle(world)\n')

    self.bu(text='Run code', command=self.run_text)
```

run_text تشبه run_file عدا أنها تأخذ النص من وشيطة نص بدلا من ملف:

```
def run_text(self):
    source = self.te_code.get(10, END)
    self.inter.run_code(source, '<user-provided code>')
```

تفاصيل تنسيق الوشائط تختلف - للأسف – باختلاف لغات البرمجة، و حتى باختلاف مديولات بايثون. تكنتر لوحده لديه ثلاثة اليات لترتيب الوشائط، تسمى هذه الاليات مهندسات المساحة geometry manager. و التي سأعرضها هنا هي grid الاخرين تسميان pack و place .

لحسن الحظ تنطبق معظم المفاهيم في هذا الفصل على المديولات الاخرى لواجهات المستخدم الرسومية و على اللغات الاخرى أيضا.

19.7 قوائم الاختيار و المستدعوات

أزرار قوائم الاختيار هي وشائط تشبه الازرار العادية، و عند الضغط عليها تبرز قائمة. و عندما تختار الفأرة عنصرا منها تختفي هذه القائمة.

التالي هو نص برمجي ينشئ زر قائمة

(يمكنك تحميله من: http://thinkpython.com/code/menubutton_demo.py) :

```
g = Gui()
g.la('Select a color:')
colors = ['red', 'green', 'blue']
mb = g.mb(text=colors[0])
```

تنشئ mb زر قائمة. مبدئيا يكون النص على الزر هو اسم اللون الافتراضي. الحلقة التالية تنشئ عنصر قائمة اختيار واحد

لكل لون:

```
for color in colors:
   g.mi(mb, text=color, command=Callable(set_color, color))
```

القرينة الاولى لـ mi هي زر القائمة الذي ترتبط به هذه العناصر.

الخيار command هو كائن يُستدعى (callable)، و هو شيئ جديد. ما رأيناه إلى الان هو استخدام طرائق مرتبطة و اقترانات لمعاودة النداء. و هي تعمل جيدا طالما أنك لا تمرر أية قرائن إلى الاقترانات. أما إن لم يكن هذا هو الحال، فعليك بناء كائن يُستدعى و يحتوي على اقتران مثل set_color، و قرائنه مثل color.

الكائنات التي تستدعى تخزن مرجعا إلى الاقتران و القرائن كخصال. ثم عندما يضغط المستخدم على زر القائمة، يقوم معاود النداء بنداء الاقتران و يمرر له القرائن التي خزنها.

هكذا ما يمكن أن يبدو عليه set_color:

```
def set_color(color):
   mb.config(text=color)
   print color
```

فعندما يختار المستخدم عنصر قائمة اختيارات و ينادى على set_color، فإنه يعد زر القائمة لإظهار اللون المختار حديثا، و يطبع كذلك اللون، إن جربت هذا المثال فسيمكنك التأكد من أن set_color نودي عندما اختير عنصر ما (و لم ينادى عندما أنشأت كائنا يُستدعى).

19.8 الربط Binding

الربط هو المشاركة بين **وشيطة** و **حدث** و **معاود نداء**: عند حدوث **الحدث** على **وشيطة** (كالضغط على زر) يتم استدعاء **معاود النداء**.

لكثير من الوشائط رابط ابتدائي. مثلا عندما تضغط على زر فإن هذا الرابط الإبتدائي يغير مظهر الزر ليبدو كأنه ضغط. و عندما ترخي الضغط فالرابط الابتدائي يسترجع المظهر الاصلي للزر ثم يستدعي معاود النداء المرتبط بخيار command لهذا الزر.

يمكنك استخدام طريقة bind لتخطي الروابط الاصلية و إضافة روابط جديدةعلى سبيل المثال، ينشئ المثال التالي رباطا لقماشة (يمكنك تحميل النص في هذا القسم من

.(http://thinkpython.com/code/draggable_demo.py

```
Ca.bind('<ButtonPress-1>', make_circle)
```

القرينة الاولى هي حدث محارف، اطلق هذا الحدث عندما ضغط المستخدم زر الفأرة الايسر أحداث الفأرة الاخرى تتضمن ButtonMotion و ButtonRelease و Double-Button.

القرينة الثانية مداولة للحدث. مداول الحدث event handler هو اقتران أو طريقة ربط، مثل معاود النداء، لكن الفرق المهم بينهما هو أن مداول الحدث يأخذ كائن Event كبرمتر. هاك مثال:

```
def make_circle(event):
   pos = ca.canvas_coords([event.x, event.y])
   item = ca.circle(pos, 5, fill='red')
```

يحتوي كائن الحدث على معلومات عن نوع الحدث و تفاصيل أخرى مثل إحداثيات مؤشر الفأرة. في مثالنا هذا المعلومات

التي نحتاج اليها هي الموقع الذي ضُغط على زر الفأرة فيه، و هذه القيم هي إحداثيات بالبكسل و يحدِّدها النظام الرسومي الذي يقبع خلفها. تُترجِم الطريقة `canvas_coords` هذه القيم إلى إحداثيات على قماشة الرسم، و هي متوافقة مع طرائق Canvas مثل `circle`.

من الشائع ربط الحدث `<Return>` بوشائط المدخِلات. و هذا الحدث يُطلَق عند الضغط على مفتاح Return أو Enter. المثال التالي يوجد زرا و مدخِلة:

```
bu = g.bu('Make text item:', make_text)
en = g.en()
enbind('<Return>', make_text)
```

ينادى على `make_text` عندما يُضغط على الزر، أو عندما يَضغط المستخدم على مفتاح `Enter` بينما يكتب في المدخلة. لكي نحقق هذا نحتاج إلى اقتران يمكن نداؤه كأمر (بدون قرائن) أو كمداول أحداث (مع حدث كقرينة):

```
def make_text(event=None):
   text = en.get()
   item = ca.text([0,0], text)
```

يحصل `make_text` على محتويات المدخلة و يعرضها كعنصر Text في القماشة.

يمكن أيضا إنشاء رابط لعناصر القماشة. التالي هو تعريف فئة لـ `Draggable` و هو فئة ابناء من `Item` تحقق لنا القدرة على الجر و الإلقاء drag-and-drop.

```
class Draggable(Item):

   def __init__(self, item):
      self.canvas = item.canvas
      self.tag = item.tag
      self.bind('<Button-3>', self.select)
      self.bind('<B3-Motion>', self.drag)
      self.bind('<Release-3>', self.drop)
```

تأخذ طريقة init عنصرا كبرمتر. و تنسخ خصاله ثم تنشئ روابط لثلاثة أحداث: ضغط الزر، حركة الزر و إرخاء الزر.

مداول الحدث `select` يخزن إحداثيات الحدث الحالي و لون العنصر الأصلي، ثم يغير اللون إلى الاصفر:

```
   def select(self, event):
      self.dragx = event.x
      self.dragy = event.y
      self.fill = self.cget('fill')
      self.config(fill='yellow')
```

`cget` هي اختصار "هاتِ الإعدادات get configuration"، تأخذ اسم الخيار كحارف و ترجع القيمة الحالية لذلك الخيار.

يحسب `drag` المسافة التي تحركها الكائن بالنسبة لنقطة البداية، و يحدّث الإحداثيات المخزنة، ثم يحرك العنصر.

```
   def drag(self, event):
      dx = event.x - self.dragx
      dy = event.y - self.dragy
      self.dragx = event.x
      self.dragy = event.y
      self.move(dx, dy)
```

الإحداثيات في هذه الحوسبة هي بالبكسل، و لا حاجة لتحويلها إلى إحداثيات القماشة.

و في النهاية تسترجع `drop` لون العنصر الأصلي:

```
    def drop(self, event):
        self.config(fill=self.fill)
```

يمكنك استخدام فئة Draggable لتزويد عنصر موجود بالقدرة على الجر و الإلقاء. فمثلا هذه نسخة معدلة من make_circle تستخدم circle لإنشاء عنصر و Draggable لجعله قابلا للجر و الإلقاء:

```
    def make_circle(event):
        pos = ca.canvas_coords([event.x, event.y])
        item = ca.circle(pos, 5, fill='red')
        item = Draggable(item)
```

هذا مثال لإحدى فوائد التوريث: يمكنك تعديل إمكانيات فئة الآباء بدون تعديل تعريفها. و هو مفيد خصوصا إن كنت ستغير سلوك مديول معرَّف لم تكتبه.

19.9 علاج الأخطاء

من التحديات التي تواجهك في برمجة GUI هي ملاحقة ما يحدث خلال بناء واجهة المستخدم، و ما يحدث كرد على أحداث المستخدم.

مثلا، من الأخطاء الشائعة عندما تقوم بإعداد معاود نداء، هو نداء الاقتران بدلا من تمرير مرجع له:

```
def the_callback():
    print 'Called.'
    g.bu(text='This is wrong!', command=the_callback())
```

إن شغلت هذا النص سترى بأنه ينادي معاود النداء أولا، **ثم** ينشئ الزر. و عندما تضغط على الزر فلن يفعل شيئا لأن القيمة المرتجعة من the_callback هي None . و في العادة عليك ألا تستدعي معاود الاتصال بينما أنت تعد واجهة المستخدم، فمعاود النداء يُستدعى فقط كرد على حدث من المستخدم.

تحد اخر في برمجة واجهات المستخدم الرسومية هو أنك لا تتحكم في سريان التنفيذ. إنها أفعال المستخدم التي تحدد أي أجزاء البرنامج ستنفذ و ما هو ترتيب هذا التنفيذ. و هذا يعني أن عليك تصميم برنامجك ليعمل بشكل صحيح عند أي تسلسل محتمل للأحداث.

مثلا، واجهة المستخدم الرسومية في التمرين 19.3 لها وشيطتان: واحدة تنشئ عنصر الدائرة و الاخرى تغير لون الدائرة. إن أنشأ المستخدم الدائرة ثم غير لونها فلا ضير. لكن ماذا لو غير المستخدم لون دائرةٍ لم توجد بعد؟ أو أنه أنشأ أكثر من دائرة؟

كلما كبر عدد الوشائط أصبح تخيل كل احتمالات تسلسل الاحداث أصعب. إحدى طرق التعامل مع هذا التعقيد هو كبسلة حالة النظام في كائن، ثم الأخذ بالاعتبار:

- ما هي الحالات المحتملة؟ في مثال الدائرة يمكننا إعتبار حالتين: قبل و بعد إنشاء المستخدم لدائرة.
- في كل حالة، ما هي الأحداث الممكن حدوثها؟ في المثال، قد يضغط المستخدم على أي من الزرين أو قد ينتهي.
- لكل زوجي حالة-حدث، ما هي المخرجات المرغوبة؟ بما أن هناك حالتين و زرين، فإن هنالك أربعة أزواج حالة-حدث للأخذ بالاعتبار.
- ما الذي قد يسبب الانتقال من حالة إلى اخرى؟ في حالتنا هذه، هناك انتقال عندما ينشئ المستخدم الدائرة الاولى.

قد تجد تعريف و اختبار اللامتغيرات مفيد، فهي يجب أن تتماسك في أي تسلسل للأحداث.

هذه المقاربة في برمجة و-م-ر ستعينك على كتابة نص صحيح من دون اضاعة الوقت في اختبار كل تسسل محتمل لأحداث المستخدم!

19.10 المعاني

و-م-ر GUI: واجهة المستخدم الرسومية graphical user interface.

وشيطة widget: إحدى العناصر التي تكون و-م-ر ، و من ضمنها الأزرار و قوائم الاختيار و حقول إدخال النصوص الخ.

خيار option: قيمة تتحكم بمظهر و وظيفة الوشيطة.

قرينة كلمة مفتاحية keyword argument: قرينة تحدد اسم البرمتر كجزء من نداء الاقتران.

معاود النداء callback: اقتران مرتبط بوشيطة يُنادى عندما يقوم المستخدم بفعلٍ ما.

طريقة مرتبطة bound method: طريقة ترتبط بتجلية محددة.

البرمجة المساقة بالاحداث event-driven programming: اسلوب في البرمجة، تسلسل التنفيذ فيه تحدده أفعال المستخدم.

حدث event: فعل من المستخدم، كالنقر على الفأرة أو الضغط على لوحة المفاتيح، يسبب ردا من و-م-ر.

حلقة انتظار الحدث event loop: حلقة لا منتهية تنتظر فعلا من المستخدم ثم ترد.

عنصر item: عنصر رسومي على وشيطة قماشة.

صندوق الاحاطة bounding box: مستطيل يحيط بمجموعة من العناصر، يحدد عادة بركنين متقابلين.

تغليف pack: ترتيب و عرض عناصر الـ و-م-ر.

هندسة المساحة geometry manager: ارتباط بين وشيطة و حدث و مداول الحدث. ينادى مداول الحدث عند حدوث الحدث في الوشيطة.

19.11 تمارين

تمرين 19.4 في هذا التمرين ستكتب عارضا للصور، هذا مثال بسيط:

```
g = Gui()
canvas = g.ca(width=300)
photo = PhotoImage(file='danger.gif')
canvas.image([0,0], image=photo)
g.mainloop()
```

يقرأ `PhotoImage` ملفا و يرجع كائن `PhotoImage` يمكن لتكنتر عرضه. يضع `Canvas.image` الصورة على القماشة، مركز الصورة هو الاحداثيات المعطاة. يمكنك وضع الصورة على الملصقات و الأزرار و الوشائط الاخرى:

```
g.la(image=photo)
g.bu(image=photo)
```

يمكن لـ *PhotoImage* التعامل مع بعض ملفات الصور مثل *GIF, PPM* لكن يمكننا استخدام مكتبة بايثون للصور *Python Image Library* (PIL) لقراءة الملفات الاخرى.

اسم مديول PIL هو image، و تكنتر به كائنا بنفس الاسم، فنتجنا للصراع سنستورد المديول مع import…as:

```
import Image as PIL
import ImageTk
```

يستورد السطر الاول Image و يمنحه الاسم المحلي PIL. السطر الثاني يستورد ImageTk الذي بوسعه ترجمة صور PIL إلى صور Tkinter PhotoImage. هذا مثال:

```
image = PIL.open('allen.png')
photo2 = ImageTk.PhotoImage(image)
g.la(image=photo2)
```

1- حمل image_demo.py و danger.py من http://thinkpython.com/code ثم شغل image_demo.py . قد تحتاج إلى تنصيب PIL و ImageTK. قد تكون من ضمن برامجك المخزنة، إن لم تكن فحملها من http://pythonware.com/products/pil.

2- في image_demo.py غير اسم الصورة الثانية من photo2 إلى photo ثم شغل البرنامج مرة أخرى. يجب أن ترى الصورة الثانية و ليس الاولى.

المشكلة هي أنك عندما تعيد تعيين photo فإنها ستتخطى المرجع إلى PhotoImage و التي ستختفي. نفس المشكلة ستحدث إن عينت PhotoImage إلى متغير محلي، فستختفي عند انتهاء الاقتران.

لتجنب هذه المشكلة عليك تخزين مرجعا لكل PhotoImage تريد الاحتفاظ بها. يمكنك استخدام متغير عمومي، أو تخزين الصور في هيكل بيانات أو كخصلة لكائن.

سلوك كهذا قد يصبح محبطا، و لهذا السبب حذرتك (و لهذا السبب أيضا الصورة العينة تقول "Danger!")

3- إنطلاقا من هذا البرنامج، اكتب يأخذ اسم المجلد و يدور في حلقة ملفاته عارضا أي ملف يتعرف عليه PIL كصورة. يمكنك استخدام عبارة try لالتقاط الملفات التي لا يتعرف عليها PIL.

4- عندما ينقر المستخدم على صورة فعلى البرنامج إظهار الصورة التالية.

يوفر PIL تشكيلة من الطرائق للتعامل مع الصور. يمكنك القراءة عنها على: http://pythonware.com/library/pil/handbook . كتحد، اختر بضعة من تلك الطرائق و اعمل و-م-ر لتطبيق الطرائق على الصور.

الحل: http://thinkpython.com/code/ImageBrowser.py .

تمرين 19.5 محرر الرسوم المتجهة هو برنامج يسمح للمستخدم برسم و تحرير الاشكال على الشاشة و انتاج ملفات الرسوم المتجهة مثل Postscript و SVG.

اكتب محررا بسيطا للرسوم المتجهية باستخدام تكنتر. يسمح على الاقل برسم الخطوط و الدوائر و المستطيلات، و يجب أن يستخدم Canvas.dump لتوليد توصيف Postscript لمحتوى القماشة.

و كتحد، بوسعك أن تمكن المستخدم من اختيار و تغيير حجم العناصر من القماشة.

تمرين 19.6 استخدم تكنتر لكتابة متصفح ويب بسيط. يجب أن يكون به وشيطة إدخال نصوص حيث يطبع المستخدم الموقع و تعرض القماشة محتويات الصفحة.

يمكنك استخدام مديول urllib لتحميل الملفات (انظر التمرين 14.6) و مديول HTMLParser لتصريف بطاقات تعريف *THML* (أنظر: https://docs.python.org/2/library/htmlparser.html .

على متصفحك التعامل مع النصوص البحتة و الارتباطات التشعبية. و كتحد، التعامل مع ألوان الخلفية و صياغة شكل نصوص البطاقات و الصور.

الملحق أ

علاج الاخطاء

قد تحدث أشكال مختلفة من الأخطاء في البرنامج. و من المهم التفريق بينها لصيدها بسرعة:

- تحدث أخطاء النحو في بايثون عندما يترجم النص المصدر إلى نص البت. و هي في العادة تعني بأن هناك خطأ في نحو البرنامج. مثلا: نسيان الفاصلة في نهاية عبارة def يصدر رسالة الخطأ (الزائدة) SyntaxError: invalid syntax .
- أخطاء عند التشغيل يظهرها المفسِّر عندما يحدث خطأ بينما يعمل البرنامج. معظم رسائل هذه الأخطاءتقول أين وقع الخطأ و أي اقتران كان ينفذ عندها. مثلا: الاجترار اللا منتهي يسبب في الاخر خطأ عند التشغيل "العمق الاقصى للإجترارات تم تعديه maximum recursion depth exceeded"
- الأخطاءالدلالية هي مشاكل البرامج التي تعمل و لم تصدر رسائل خطأ، لكنها لا تقوم بالعمل المطلوب. مثلا: تعبير لم يقيم بالشكل الذي تريده ينتج نتيجة غير صحيحة.

الخطوة الاولى في علاج الأخطاء هي أن تعرف أي نوع من الأخطاء تواجه. رغم كون الاقسام التالية مرتبة حسب نوع الخطأ إلا أن بعضها يطبق في عدد من الظروف .

أ-1 الأخطاء النحوية

يَسهُل إصلاح هذه الأخطاء في العادة عندما تعرف ما هو الخطأ. و للأسف فرسائل الخطأ ليست واضحة دائما. أكثر الرسائل شيوعا هي SyntaxError: invalid syntax و SyntaxError: invalid token. و ليس أي منها تفيدنا بمعلومات كافية.

لكن على الجانب الاخر، فالرسالة تخبرك أين وقعت المشكلة. الحقيقة أنها تخبرك أي لاحظ بايثون المشكلة، و هو ليس بالضرورة مكان الخطأ. أحيانا يكون الخطأ سابقا لمكان الرسالة، و في الغالب السطر السابق.

ان كنت قد بنيت البرنامج بالتدريج فيجب أن يكون لديك فكرة عن مكان وجود الخطأ، سيكون في آخر سطر كتبته.

ان كنت قد نسخت النص من كتاب ابدأ بمقارنة النصين بعناية، افحص كل حرف. في نفس الوقت تذكر بأن الخطأ قد يكون في الكتاب، فإن رأيت فيه ما يظهر كأنه خطأ نحوي فقد يكون خطأ نحوي فعلا.

هنا بعض الطرق التي تجنبك الأخطاءالنحوية:

1- تأكد من أنك لا تستعمل كلمات بايثون المفتاحية كمتغيرات.

2- تأكد من وجود نقطتان في نهاية ترويسة العبارات المركبة، مثل for, while, if, def.

3- تأكد من وجود علامات الاقتباس قبل و بعد المحارف.

4- ان كان لديك محارفا متعددة السطور مع علامات الاقتباس الثلاثة (مفردة أو مزدوجة) تأكد من أنك أغلقت المحارف بالشكل الصحيح. المحارف غير المقفلة تسبب invalid token في نهاية برنامجك، أو أنها ستعامل الاجزاء اللاحقة في البرنامج كمحارف إلى أن تصل إلى المحارف التالية. و في الحالة الثاني قد لا تصدر رسالة خطأ على الاطلاق!!

5- الاقواس الغير مقفلة } ,) ,] تتسبب في جعل بايثون يستمر إلى السطر التالي على أنه جزء من العبارة. في العموم يصدر الخطأ في السطر التالي لحظيا تقريبا.

6- انتبه للخطأ التقليدي = بدلا من == في المشروطات.

7- انتبه للمسافات البادئة و أنها مصطفة بالشكل الذي يجب أن تكون عليه. بايثون يستطيع التعامل مع الفراغات و مسافات الجدولة لكن إن خلطتها فستتسبب بالمشاكل. أفضل طريقة لتجنب هذه الأخطاء هي استعمال محرر نصوص يعلم عن بايثون و يدرج المسافات البادئة الصحيحة.

إن لم ينفع أي من السابق فانتقل إلى القسم التالي....

أ-1-1 أغير و أبدل لكن لا شيء يتغير.

إن قال المفسِّر أن هناك خطأ في النص لكنك لا ترى أين، فقد يعني أنكما (أنت و المفسّر) لا تنظران إلى نفس النص. افحص في بيئة البرمجة لديك إن كان النص الذي تحرره هو النص الذي يحاول بايثون تشغيله.

إن لم تكن متأكدا فتعمد وضع خطأ في بداية النص. الآن شغل البرنامج مرة ثانية. إن لم يجد المفسِّر الخطأ الجديد فإنه يشغل ملفا اخر.

المتهمون المحتملون هنا قليلون:

- أنت، لقد عدلت على النص و لم تحفظ الملف قبل التشغيل. بعض بيئات البرمجة تقوم بهذا عنك، لكن بعضها لا.
- أنت، غيرت اسم الملف لكنك لازلت تشغل الملف ذو الاسم القديم.
- بيئة البرمجة، هناك شيء لم يُعَدّ بالشكل الصحيح.
- إن كنت تكتب مديول، إنتبه لئلا تعطيه اسما موجودا لأحد مديولات بايثون.
- ان استوردت بـ import لقراءة مديول فتذكر بأن عليك أعادة تشغيل المفسّر أو ستخدام reload لقراءة ملف معدل. إن استوردت المديول ثانية فإنه لا يفعل أي شيء.

إن علقت و لم تستطع إكتشاف ما الذي يحدث، فهناك مقاربة أخرى، و هي البدء بشيء جديد مثل برنامج "Hello, World!" للتأكد من أنه بإمكانك جعل برنامج معروف يعمل. ثم ابدأ بإضافة أجزاء برنامجك بالتدريج إلى البرنامج الجديد.

أ-2 أخطاء عند التشغيل

عندما يكون برنامجك صحيح نحويا فسيتمكن بايثون من تشغيله، أو البدء في تشغيله. فما الذي يمكن أن يخفق الان؟

أ-1-1 برنامجي يقوم بلا شيء البتة.

تشيع هذه المشكلة عندما يتكون برنامجك من فئات و اقترانات لكنه لا يستدعي شيئا لبدء التنفيذ. قد تتعمد هذا متعمدا إن كنت تخطط لاستيراد هذا المديول للتزود بالفئات و الاقترانات.

إن لم يكن هذا هو المقصود، فتأكد أنك تستدعي اقترانا لبدء التنفيذ، أو نفذ اقترانا في الوضع التفاعلي. أيضا نظر قسم "سريان التنفيذ" اللاحق.

أ-2-2 برنامجي يعلق

إن توقف برنامج و ظهر بأنه لا يقوم بشيء فهو عالق hanging. كثيرا ما يعني هذا أنه علق في حلقة لا منتهية أو إجترار لا منتهي.

- إن كانت هناك حلقة ما تشك فيها فأضف عبارة `print` قبلها مباشر و لتقل فيها "ها أنا أدخل الحلقة" و عبارة بعد الحلقة مباشرة و لتقل "ها أنا أخرج من الحلقة".

  شغل البرنامج و إن رأيت "ها أنا أدخل الحلقة" و لم تر "ها أنا أخرج من الحلقة" فأنت في حلقة غير منتهية، إن حدث هذا فانتقل إلى القسم "الحلقة اللا منتهية".

- في معظم الاحيان يتسبب الاجترار اللا منتهي في جعل البرنامج يعمل لفترة ثم يصدر رسالة: `RuntimeError: Maximum recursion depth exceeded`. إن حدث هذا فانتقل إلى القسم "الاجترار اللامنتهي" اللاحق.

- إن لم تعمل أي من هاتين الطريقتين فابدأ باختبار الحلقات و الاجترارات الاخرى و كذلك الطرائق.

- إن لم يحل هذا المشكلة فقد يعني أنك لا تفهم سريان التنفيذ لبرنامجك. إذن فاذهب إلى القسم "سريان التنفيذ" اللاحق.

حلقة لا منتهية

إن كنت تظن أن لديك حلقة لا منتهية و تظن بأنك تعلم أي منها التي تسبب المشكلة فأضف عبارة `print` في نهايتها لتطبع قيم المتغيرات في المشروطة و قيمة المشروطة.

كمثال:

```
while x > 0 and y < 0 :
   # do something to x
   # do something to y

   print "x: ", x
   print "y: ", y
```

```
print "condition: ", (x > 0 and y < 0)
```

الان، عندما تشغل البرنامج سترى ثلاثة سطور من المخرجات لكل لفة في الحلقة. و اخر لفة في الحلقة سيكون الشرط False . أما إن استمرت الحلقة بالعمل فسترى قيم x و y و قد تكتشف السبب الذي يجعلهما لا يتحدثان بالشكل الصحيح.

اجترار لا منتهي

في معظم الاحيان يتسبب الاجترار اللا منتهي في جعل البرنامج يعمل لفترة ثم يصدر رسالة: RuntimeError: Maximum recursion depth exceeded.

إن شككت بأن اقترانا أو طريقة ما هي ما يسبب الاجترار اللا منتهي، فابدأ البحث للتأكد من وجود حالة الاساس. بكلمات أخرى: يجب أن يكون هناك شرط ما يجعل الاقتران أو الطريقة ترجع دون التسبب في استدعاء اجتراري. إن لم يكن فعليك إعادة التفكير في الخوارزمية و أن تحدد حالة أساس.

إن كانت هناك حالة أساس لكن البرنامج لا يصل إليها فأضف عبارة print بداية الاقتران أو الطريقة و لتطبع البرمترات. الان عندما تشغل البرنامج سترى سطورا من المخرجات في كل مرة يُستدعى فيها الاقتران أو الطريقة، إن كانت البرمترات لا تتطور نحو الحالة الاساس فسيكون لديك فكرة عما يحدث.

سريان التنفيذ

إن لم تكن متأكدا من كيفية سير التنفيذ في برنامجك، أضف print في بداية كل اقتران و لتطبع شيئا كـ " entering function foo" حيث foo هو اسم الاقتران.

الان عندما تشغل البرنامج ستطبع ملاحقة لكل اقتران تم استدعاؤه.

أ-3-3 عندما أشغل البرنامج أحصل على استثناء

إن حدث خطأ خلال التشغيل سيطبع بايثون رسالة تحتوي اسم الاستثناء و السطر الذي حدثت عنده المشكلة و ملاحقة.

الملاحقة تسمّي الاقتران الذي كان ينفذ حينها، و الاقتران الذي استدعى هذا الاقتران، و الاقتران الذي استدعاه و هكذا. بكلمات اخرى فهي تلاحق تسلسل استدعاءات الاقترانات التي أوصلت البرنامج إلى هذه الحال. تحتوي الملاحقة أيضا على رقم السطر الذي حدث فيه النداء.

الخطوة الاولى هي فحص مكان حدوث الخطأ في البرنامج و محاولة التعرف على المشكلة هناك. هنا بعض الأخطاءالشائعة التي تحدث عند التشغيل:

خطأ في التسمية NameError: أنت تحاول استخدام متغير غير موجود في البيئة.

تذكر أن المتغيرات المحلية هي محلية. فلا يمكنك الاشارة اليها من خارج الاقتران الذي عرفت فيه.

خطأ في النمط TypeError: هنالك عدة أسباب محتملة:

- أنت تحاول استخدام قيمة بطريقة خاطئة. مثال: استخدام شيء غير الاعداد الصحيحة كمؤشرات في القوائم، المحارف أو التوبل.
- هناك عدم تطابق بين العناصر في محارف صيغة و العناصر التي مررت للتحويل. قد يحدث هذا إما لعدم تطابق عدد العناصر أو النداء على تحويل غير صالح.
- أنت تمرر العدد الخطأ من القرائن لاقتران أو طريقة. للطرائق، أنظر في تعريف الطريقة للتأكد من أن البرمتر الاول هو `self`. ثم انظر إلى استدعاء الطريقة للتأكد أن استدعاءها كان على كائن له النمط الصحيح و يزود القرائن الصحيحة.

أخطاء المفاتيح KeyError: أنت تحاول الوصول إلى عنصر في القاموس باستخدام مفتاح لا يحتويه القاموس.

أخطاء الخصال AttributeError: أنت تحاول الوصول إلى خصلة أو طريقة غير موجودة. تأكد من التهجئة!! يمكنك استخدام `dir` لسرد جميع الخصال الموجودة.

إن كان الخطأ يقول `NoneType` فهذا يعني أنه `None` . أحد الاسباب الشائعة هو نسيان إرجاع القيمة من الاقتران، إن وصلت إلى نهاية الاقتران دون أن تصادف عبارة `return` فسوف ترجع `None` , و سبب شائع اخر هو استخدام النتيجة من طرق القوائم مثل sort و التي ترجع None.

خطأ المؤشر IndexError: المؤشر الذي تستخدمه للوصول إلى عنصر في قائمة أو محارف أو توبل أكبر من طولها ناقص واحد. أضف عبارة print قبل موقع الخطأ مباشرة، و لتطبع قيمة المؤشر و طول المصفوفة. هل المصفوفة بالطول الصحيح؟ هل المؤشر بالقيمة الصحيحة؟

من المجدي استخدام معالج أخطاء بايثون (`pdb`) لأنه يلاحق الاستثناءات و يسمح لك برؤية حالة البرنامج قبل الخطأ. يمكنك القراءة عن pdb على: `http://docs.python.org/2/library/pdb.html`.

أ-2-4 أضفت الكثير من عبارات `print` إلى أن غمرتني المخرجات

من مشاكل استخدام `print` في علاج الأخطاء هي أنك تكتشف نفسك مطمورا بالمخرجات. هناك طريقتان لتكمل علاج برنامجك: بسط المخرجات أو بسط البرنامج.

لتبسيط المخرجات، يمكنك حذف عبارات `print` التي لا تساعد أو التعليق عليها، أو يمكنك تجميعها معا أو تغيير تنسيقها فتصبح أسهل للفهم.

لتبسيط البرنامج، هناك العديد من الأشياء التي يمكنك فعلها. أولها تخفيض حجم المسألة التي يعمل عليها البرنامج. مثلا إن كان يبحث في قائمة، صغر له حجمها. إن كان البرنامج يأخذ مدخلا من المستخدم، إعطه أبسط مدخل يسبب المشكلة.

ثانيا تنظيف البرنامج. أزل النصوص الميتة و أعد ترتيب البرنامج بشكل يسهل معه فهمه. مثلا إن كنت تشك بأن المشكلة موجودة في جزء عشي من البرنامج حاول إعادة كتابة الجزء العشي و تبسيط بنائه. إن كنت تشك في اقتران ضخم حاول تجزئته إلى اقترانات أصغر و فحصها متفرقة.

في الغالب تكون عملية العثور على حالة الفحص الصغرى هي ما تدلك على البقة. إن لاحظت بأن البرنامج يعمل في مواقف و لا يعمل في اخرى، فاستخدم هذا كدليل لما يحدث.

و في نفس السياق فإعادة كتابة جزء من النص يساعد في العثور على البقات الدقيقة. فإن قمت بتغيير تعتقد أنه لن يؤثر على

البرنامج إلا أنه أثر فهذا تلميح اخر تستفيد منه.

أ-3 الأخطاء الدلالية

علاج الأخطاء الدلالية أصعب مني بعض النواحي، ذلك لأن المفسِّر لا يزودك بمعلومات عما يحدث. أنت فقط من يعلم ما على البرنامج القيام به.

الخطوة الاولى أن تربط بين نص البرنامج و السلوك الذي تراه. يجب أن تكون لديك فرضية عما يحدث من البرنامج فعليا. و مما يصعب هذه المهمة هي أن الحواسيب سريعة.

قد ترغب أحيانا في ابطاء البرنامج إلى سرعة بشرية. بعض معالجات الأخطاءتفعل ذلك. لكن الوقت المستنفد في إدراج بعض عبارات `print` يكون في الغالب أقل من ذلك المطلوب لإعداد معالج أخطاء، و إدراج و حذف نقاط الفحص و تسيير البرنامج بخطوات إلى مكان حدوث المشكلة.

أ-1-1 برنامجي لا يعمل.

عليك أن تسأل نفسك هذه الاسئلة:

- هل هناك شيء مما يفترض من البرنامج عمله لكنه لا يحدث؟ اعثر على الفقرة من النص و التي ينفذ فيها ذلك الاقتران و تأكد من أنها تنفذ في عندما يجب أن تنفذ.
- هل هناك شيء يحدث و يفترض ألا يحدث؟ اعثر على الفقرة في البرنامج التي تقوم بذلك الاقتران و تأكد من أنها تنفذ فقط عندما يجب أن تنفذ.
- هل هناك فقرة في البرنامج تقوم بتأثير غير الذي تتوقعه؟ تأكد من أنك تفهم النص المقصود، خصوصا إن كان به استدعاءات إلى اقترانات أو طرائق من مديولات بايثون الاخرى. اقرأ وثائق الاقتران الذي تستدعيه. جربه بكتابة حالة فحص بسيطة و افحص النتائج.

لكي تبرمج يجب أن يكون لديك نموذج عقلي للطريقة التي تعمل بها البرامج. فإن كتبت برنامجا لا يقوم بما تريده فالمشكل في كثير من الاحيان ليست في البرنامج، بل في النموذج العقلي الذي بنيته.

أفضل طريقة لتصحيح نموذجك العقلي عن البرنامج هي تقسيمه إلى مكونات (عادة الاقترانات و الطرائق) و فحص كل مكون على حدة. و بمجرد عثورك على التناقض بين النموذج و الواقع ستتمكن من حل المشكلة.

طبعا يجب أن تبني و تختبر بينما تصمم برنامجك. إن واجهت مشكلة فسيكون حجم ما عليك فحصه من النصوص صغيرا و هي في الغالب الاضافات الجديدة.

أ-1-2 لدي تعبير مشعراني كبير و لا يقوم بما أتوقعه منه.

لا بأس في كتابة التعبيرات الكبيرة طالما أن قراءتها سهلة، لكنها تكون صعبة العلاج من الاخطاء. التفكير السليم هو تقسيم التعبيرات المعقدة إلى مجموعة من التعيينات إلى متغيرات مؤقتة.

مثلا:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

يمكن كتابته هكذا:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

النسخة المسهبة اسهل للقراءة لأن أسماء المتغيرات تضيف إلى المعلومات، و هي اسهل للعلاج لأنك تستطيع فحص أنماط المتغيرات الوسيطة و قيمها.

مشكلة اخرى للتعبيرات الضخمة هي أن تراتب تقييمها قد لا يكون ما تتوقع. مثلا إن كنت تترجم التعبير $x/2\pi$ إلى بايثون فقد تكتب:

```
y = x / 2 * math.pi
```

و هذا غير صحيح لأن الضرب و القسمة لهما نفس الاسبقية و تقيم من اليسار إلى اليمين. إذن فهذا التعبير سيقيم على أنه $x\pi/2$.

أفضل طريقة لعلاج التعبيرات هي استخدام الاقواس لجعل ترتيب التقييم واضحا:

```
y = x / (2 * math.pi)
```

فأينما كنت غير متأكد من ترتيب التقييم استخدم الاقواس. ليس فقط ليكون البرنامج صحيحا (يقوم بما تريد) و لكن لجعله مقروءا لأولئك الذين لم يحفظوا قوانين الاسبقية.

أ-3-3 لدي اقتران أو طريقة لا ترجع ما أتوقعه منها.

إن كانت لديك عبارة return في تعبير معقد فلن يكون لديك الوقت لطباعة قيمة return قبل الارجاع. مرة أخرى يمكنك استخدام المتغيرات المؤقتة. مثلا، بدلا من:

```
return self.hands[i].removeMatches()
```

يمكنك كتابة:

```
count = self.hands[i].removeMatches()
return count
```

هكذا تكون لديك الفرصة لطباعة قيمة count قبل الارجاع.

أ-3-4 أنا في ورطة حقيقية و أريد المساعدة.

أولا حاول الابتعاد عن الحاسوب لعدة دقائق. الحواسيب تصدر أمواج تؤثر في المخ و تسبب هذه الأعراض:

- الإحباط و الغضب.
- الإيمان بالخرافات "هذا الحاسوب يكرهني" و التفكير السحري ("إنه يعمل عندما ارتدي قبعتي بالعكس").
- برمجة الخطى العشوائية (محاولة كتابة كل البرامج المحتملة و اختيار البرنامج الذي يقوم بالعمل الصائب).

إن لاحظت أنك تعاني إحدى هذه الاعراض قف و اذهب في مشوار. و عندما تهدأ فكر في البرنامج. ما الذي يفعله؟ ما هي الاسباب المحتملة لسلوكه هكذا؟ متى كان البرنامج يعمل بشكل صحيح و ما الذي فعلته بعدها؟

أحيانا العثور على البق يستغرق الوقت. كثيرا ما أعثر على الأخطاء بينما أنا بعيد عن الحاسوب، عندما أجعل دماغي يسرح. من أفضل الاماكن للعثور على البق هي القطارات، الدش، و السرير قبل النوم.

أ-5-3 لا، أنا فعلا بحاجة للمساعدة.

هذه أمور تحدث. حتى أفضل المبرمجين له كبوات. أحيانا تعمل على برنامج لفترة طويلة بما فيه الكفاية لتجعلك لا ترى الخطأ. استخدام زوجي عيون طازجة هو الحل.

قبل أن تطلب المساعدة من أحد تأكد من أنك مستعد. برنامجك يجب أن يكون بسيطا قدر الإمكان، و يجب أن تكون منهمكا بفحصه بأبسط مدخل ممكن للتسبب بالخطأ. يجب أن تكون قد وضعت عبارات `print` في الامكنة المناسبة (و ما تطبعه هذه العبارات يجب أن يكون مفهوما). يجب أن تكون فاهما للمشكلة بالقدر الذي يسمح لك بشرحها بوضوح.

و عندما تستشير أحدهم تأكد من إعطائه المعلومات التي سيحتاجها:

- إن كانت هناك رسائل خطأ، فما هي و بأي جزء من البرنامج تتعلق؟
- ما هو اخر شيء قمت به قبل وقوع الخطأ؟ ماذا كانت آخر السطور التي كتبتها قبله، أو ما هي حالة الاختبار الجديدة التي فشلت.
- ما الذي جربته لحلها حتى الان، و ماذا تعلمت منه؟

و عندما تجد البقة، فكر لثوان بالذي كان يجب فعله للعثور عليها أسرع. ففي المرة التالية التي ترى فيها شيئا مماثلا ستتمكن من الامساك بها أسرع.

تذكر بأن الهدف ليس جعل البرنامج يشتغل. بل التعلم كيف تجعل البرنامج يشتغل.

الملحق ب

تحليل الخوارزميات

هذا الملحق هو اقتباس محرَّر من "فكِّر تعقيدات" لـ ألن ب. داوني، من منشورات أوريلي ميدا 2011 . عند انتهائك من هذا الكتاب قد تود الانتقال إلى ذاك.

تحليل الخوارزميات هو فرع من علوم الحاسوب يبحث في أداء الخوارزميات، خصوصا زمن تشغيلها و المساحة المتطلبة لها. أنظر http://em.wikipedia.org/wiki/Analysis_of_algorithms.

الهدف العملي من تحليل الخوارزميات هو توقع أداء الخوارزميات المختلفة للإستدلال عند اصدار القرارات في مرحلة التصميم.

في الحملة الرئاسية لعام 2008 عندما زار المرشح باراك أوباما جوجل، طُلب منه أن يقوم بتحليل مرتجل. سأله كبير التنفيذيين إريك شمدت مازحا عن "أكثر الطرق فعالية لفرز مليون من الاعداد الصحيحة 32 بت"، على ما يبدو أن أحد ما غشش أوباما لأنه أجاب بسرعة "أعتقد أن **فرز الفقاعة** سيكون الطريقة الخطأ أيضا" أنظر:

http://www.youtube.com/watch?v=k4RRi_ntQc8.

كانت تلك اجابة صحيحة: فرز الفقاعة من ناحية المفهوم بسيط لكنه بطيء في حالة مجموعات البيانات الضخمة. الجواب الذي كان يتوقعه شمدت ربما كان "خوارزمية فرز التراتب"[1].

http://en.wikipedia.org/wiki/Radix_sort.

الهدف من تحليل الخوارزميات هو المقارنة ذات المغزى بين الخوارزميات، لكن هناك بعض المشاكل:

- قد يعتمد الأداء النسبي للخوارزمية على خصائص الحاسوب (أو الالة)، فقد تكون خوارزمية ما أسرع على الالة **أ**، و خوارزمية أخرى أسرع على الالة **ب**. الحل العام لهذه المشكلة هو تحديد الالة النموذج ثم تحليل عدد الخطوات، أو العمليات التي تتطلبها الخوارزمية تحت النموذج المعطى.
- قد يعتمد الأداء النسبي على تفاصيل مجموعة البيانات. مثلا بعض خوارزميات الفرز تعمل أسرع إن كانت البيانات مفروزة جزئيا، خوارزميات أخرى تعمل أبطأ. الطريقة الشائعة لتجنب هذه المشكلة هو تحليل "سيناريو أسوأ الحالات". يفيد أحيانا أن نحلل الأداء على الحالة المتوسطة، إلا أن هذا في العادة أصعب، و قد لا يكون واضحا أي حالة يمكن أن تصنف كمتوسطة.
- يعتمد الأداء النسبي أيضا على حجم المشكلة. فخوارزمية فرز سريعة على القوائم القصيرة، قد تصبح بطيئة على القوائم الطويلة. الطريقة المعتادة للتعامل مع هذه المشكلة هي التعبير عن زمن التشغيل (أو عدد العمليات) كاقتران لحجم المشكلة، ثم مقارنة الاقترانات بالمقارنة asymptotically كلما زاد حجم المشكلة.

([1] أظن أن أفضل إجابة لهذا السؤال هي "أفضل طريق لترتيب مليون عدد صحيح هي استخدام أي طريقة فرز تعطيني إياها اللغة التي أستخدمها. فأداؤها مقبول لمعظم الغايات، و إن كانت بطيئة سأستخدم أداة لاكتشاف موقع استنفاذ الوقت. إن ظهر بأن طريقة الفرز الأسرع لها تأثير ملموس على الأداء فسأبحث عن طريقة جيدة للفرز الترتيبي")

الجيد في هكذا مقارنات هي أنها يمكن أن تستخدم في تبسيط تصنيف الخوارزميات. مثلا، إن كنتُ أعلم بأن زمن التشغيل للخوارزمية **أ** يتناسب مع حجم مدخلاتٍ قيمته **ن** و زمن تشغيل الخوارزمية **ب** يتناسب مع **ن**٢، عندها يمكنني توقع أن تكون الخوارزمية **أ** أسرع من **ب** لقيم **ن** الكبيرة.

يأتي هذا النوع من التحاليل مع بعض التحذير لكننا سنأتي على هذا فيما بعد.

ب-1 تراتب النمو

إفترض أنك حللت خوارزميتان و عبرت عن زمن التشغيل لهما بحجم المدخل: الخوارزمية **أ** تأخذ 100**ن** + 1 خطوة لتحل مسألة بحجم **ن**، الخوارزمية **ب** تأخذ **ن**٢ + **ن** + 1 خطوة.

الجدول التالي يظهر زمن التشغيل لهذه الخوارزميات لأحجام مختلفة من المسائل:

حجم المدخلات **ن**	زمن التشغيل للخوارزمية **أ**	زمن التشغيل للخوارزمية **ب**
10	1001	111
100	10001	10101
1000	100001	1001001
10000	1000001	$>10^{10}$

عندما كانت **ن** = 10 ظهرت الخوارزمية **أ** بمظهر سيئ، فهي تأخذ عشرة أضعاف الوقت الذي تأخذه الخوارزمية **ب**. لكن عندما كانت **ن** = 100 أصبحتا متساويتين تقريبا ، لكن للقيم الاكبر كانت **أ** أفضل بكثير.

السبب الرئيسي هو أنه لقيم **ن** الكبيرة، تكبر الاقترانات التي تحتوي على **ن**٢ على نحوٍ أسرع من الاقتران الذي حده القائد **ن**، الحد القائد (leading term) هو الحد الذي أسه الأكبر.

كان معامل الحد القائد في الخوارزمية **أ** كبير (100) لذلك كان أداء الخوارزمية **ب** أفضل لقيم **ن** الصغيرة. لكن بغض النظر عن المعامل، سيكون هناك دائما قيمة لنون حيث **أ** × **ن** ^{2} > **ب** × **ن**.

البرهان نفسه ينطبق على الحدود غير القائدة. حتى لو كان زمن التشغيل للخوارزمية **أ** هو **ن** + 1000000 فستظل أفضل من الخوارزمية **ب** لقيم **ن** الكبيرة كفاية.

نحن في العموم نتوقع من الخوارزمية التي لها حد قائد صغير أن تكون لها الافضلية في المشاكل الكبيرة، لكن للمشاكل الصغيرة فلا بد و أن تكون هناك الشعرة التي يكون على جانبها الاخر أن الخوارزمية الثانية هي الافضل. موقع هذه الشعرة يعتمد على تفاصيل الخوارزمية، و على المدخلات و كذلك على الالة. و لهذا السبب تهمل في غايات تحليل الخوارزميات. لكن لا يعني هذا أن عليك نسيانها.

إن كان لخوارزميتين نفس الحد القائد، فسيصعب القول أيها أفضل، مرة أخرى يعتمد القرار على التفاصيل. لذا و لغرض التحليل الخوارزمي تعتبر الاقترانات التي لها نفس الحد القائد متكافئة، حتى و إن لم تكن لها نفس العوامل.

ترتيب النمو هو مجموعة من الاقترانات التي يعتبر سلوك نموها العرضي متكافئا. مثلا 2**ن** و 100**ن** و **ن**+1 كلها تنتمي إلى نفس تراتب النمو. و يكتب هكذا $O(n)$ و غالبا ما تسمى "خطية" لأن كل اقتران في المجموعة ينمو خطيا بنمو n.

كل الاقترانات ذات الحد القائد n^2 تنتمي لـ $O(n^2)$ فهي تربيعية (الاسم الذي يطلق على الاعداد التي أسها 2).

نرى في الجدول التالي بعض تراتيب النمو التي تظهر بشكل شائع في التحليل الخوارزمي، بترتيب تصاعدي من حيث السوء.

Order of growth	Name	الاسم
$O(1)$	constant	ثابت
$O(\log_b n)$	logarithmic (for any b)	لوغرتمي (لأي b)
$O(n)$	linear	خطي
$O(n \log_b n)$	"en log en"	"en log en"
$O(n^2)$	quadratic	تربيعي
$O(n^3)$	cubic	تكعيبي
$O(c^n)$	exponential (for any c)	أسي (لأي c)

بالنسبة للحدود اللوغرتمية فأساس اللوغرتم لا يهم، تغيير الأسس كالضرب بثابت. و هو لا يغير تراتب النمو. و نفس الأمر ينطبق على الاقترانات الاسية، كل الاقترانات الاسية تنتمي إلى نفس ترتيب النمو بغض النظر عن قاعدة الأس. تنمو الاقترانات الاسية بسرعة، لذلك فالاقترانات الاسية مفيدة في حل المشاكل الصغيرة فقط.

تمرين ب-1 اقرأ صفحة ويكيبيديا عن Big-Oh على

`http://en.wikipedia.org/wiki/Big_O_notation`. ثم أجب عن الاسئلة التالية:

1- ماهو ترتيب النمو لـ $n^3 + n^2$؟ ماذا عن $1000000n^3+n^2$؟ و عن $n^3 + 1000000n^2$؟
2- ماهو ترتيب النمو لـ $(n^2 + n).(n+1)$؟ قبل أن تبأ بالضرب تذكر أن كل ما تحتاجه هو الحد القائد.
3- إن كانت f في $O(g)$ لأي إقتران غير محدد g، ماذا نستطيع القول عن $af + b$؟
4- إن كانت $f1$ و $f2$ في $O(g)$ ماذا نقول في $f1 + f2$؟
5- إن كانت $f1$ في $O(g)$ و $f2$ في $O(h)$، فماذا نقول في $f1 + f2$؟
6- إن كانت $f1$ في $O(g)$ و $f2$ في $O(h)$، فماذا نقول في $f1 . f2$؟

هذا النوع من التحليل غالبا ما يجده المبرمجون الذين يهتمون بالاداء عسيرا على الهضم. و لهم بعض الحق في ذلك: أحيانا لا يَفرِق المعامل عن الحد غير القائد. فأحيانا تكون تفاصيل الالة و لغة البرمجة و خصائص المدخلات هي التي تصنع الفرق الأكبر. و للمشاكل الصغيرة فالسلوك التقاربي ليس ذا تأثير.

لكن إن أبقيت هذه التحذير في بالك، سيكون التحليل الخوارزمي أداة مفيدة، على الاقل للمشاكل الكبيرة. فالخوارزميات "الأفضل" هي في الغالب "أفضل"، و أحيانا " أفضل بكثير". الفرق بين خوارزميتين جيدتين لهما نفس ترتيب النمو يكون في العادة عامل ثابت، لكن الفرق بين خوارزمية جيدة و خوارزمية سيئة لا يتسع له مكان!

ب-2 تحليل عمليات بايثون الأساسية

معظم العمليات الحسابية ثابتة الوقت، الضرب يأخذ في العادة وقتا أطول من الجمع، و القسمة تأخذ وقتا أطول من كليهما، لكن زمن التشغيل هنا لا يعتمد على علو قيمة العوامل. يُستثنى من هذا الارقام الكبيرة جدا، فهناك يطول زمن التشغيل بزيادة عدد الخانات.

عمليات الفهرسة – قراءة أو كتابة عناصر تسلسل أو قاموس – أيضا ثابتة الوقت، بغض النظر عن حجم هيكل البيانات.

حلقة `for` التي تدور في تسلسل أو قاموس، تكون في العادة خطية طالما كانت كل العمليات في متن الحلقة ثابتة الوقت. مثلا ضم عناصر قائمة هو خطي:

```
total = 0
for x in t:
     total += x
```

الاقتران الجاهز `sum` خطي لأنه يفعل نفس الشيء، لكنه أسرع على نحو ما بسبب أنه تطبيق أكثر فعالية، و في لغة التحليل الخوارمي: له معامل قائد أصغر.

ان استخدمت نفس الحلقة "لإضافة" قائمة من المحارف، فزمن التشغيل يكون تربيعي لأن إضافة المحارف خطي.

طريقة `join` أسرع لأنها خطية على مدى الطول الكلي للمحارف.

و بحكم الخبرة، إذا كان متن حلقة في $O(n^a)$ فإن الحلقة ككل تكون في $O(n^{a+1})$. الاسثناء هو إن أثبَتَّ أن الحلقة تظل موجودة بعد عدد ثابت من التكرارات. إن اشتغلت الحلقة عدد k من المرات بغض النظر عن n ، فإنها تكون في $O(n^a)$ حتى عندما تكون k كبيرة.

الضرب في k لا يغير تراتب النمو، لكن هذا لا يختلف عن القسمة. لذا فإن كان متن الحلقة في $O(n^a)$ و اشتغلت عدد مرات يساوي n/k، فالحلقة تكون في $O(n^{a+1})$، حتى لـ k الكبيرة.

معظم عمليات المحارف و التوبلات خطية، لكن إن كانت أطوال المحارف محدودة بثابت، كالعمليات على حرف واحد، فسوف تعتبر ثابتة الوقت.

معظم عمليات القوائم خطية، إلا هذه الاستثناءات:

- إضافة عنصر في نهاية قائمة هو ثابت الوقت في المتوسط، عندما لا يظل متسعا فإنها تُنسخ إلى موقع أكبر، لكن الوقت الإجمالي لعدد n من العمليات هو $O(n)$ ، لهذا نقول إن الوقت المحفوظ (لسداد الدين) imortized لعملية واحدة هو $O(1)$.
- حذف عنصر من نهاية القائمة هو ثابت الوقت.
- الفرز هو $O(n \log n)$

معظم عمليات القواميس و طرائقها ثابتة الوقت، لكن هناك استثناءت أيضا:

- زمن التشغيل لـ `copy` يتناسب مع عدد العناصر، لكن ليس مع حجم العناصر (فهي تنسخ مراجع و ليس العناصر نفسها).
- وقت التشغيل لـ `update` يتناسب مع حجم القاموس الذي مُرر كبرمتر، و ليس القاموس الذي يجري تحديثه.
- `Keys, values, items` خطية لأنها ترجع قوائم جديدة، و `Iterkeys` و `itervalues` و

iteritems ثابتة الوقت لأنها ترجع تكرارات. لكن حلقة التكرارات فالحلقة تكون خطية. استخدام اقتران iter يخفض بعض النفقات لكنه لا يغير تراتب النمو إلا إذا كان عدد العناصر التي تعالجها محدد.

- أداء القواميس هو أحد معجزات علوم الحاسوب، سنرى كيف تعمل في القسم ب-4.

تمرين ب-2 إقرأ صفحة ويكيبيديا عن الخوارزميات على

http://en.wikipedia.org/wiki/Sorting_algorithm ثم أجب عن الاسئلة التالية:

1- ما هو "الفرز بالقياس"؟ ما هي أفضل أسوأ-حالة ترتيب النمو للفرز بالقياس؟ ما هي أفضل أسوأ-حالة ترتيب النمو لأي خوارزمية فرز؟
2- ما هو ترتيب النمو لفرز الفقاعة، و لماذا يظن باراك أوباما بأنها الطريقة الخطأ؟
3- ما هو ترتيب النمو لفرز الاساس؟ ما هي الشرط المسبقة لاستعماله؟
4- ما هو الفرز المستقر و لماذا هو مهم عمليا؟
5- ما هي أسوأ خوارزمية فرز (و يكون لها اسم)؟
6- أي خوارزمية فرز تستخدم مكتبة C؟ و أي خوارزمية فرز تستخدم مكتبة بايثون؟ هل هذه الخوارزميات مستقرة؟ قد تتطلب اللإجابة منك أن تبحث في جوجل.
7- كثير من خوارزميات الفرز اللامقارن خطية، إذن فلماذا يستخدم بايثون O(n log n)؟

ب-3 تحليل خوارزميات البحث

البحث هو خوارزمية تأخذ مجموعة و عنصر مستهدف، و تقرر فيما إذا كان الهدف في المجموعة، و غالبا ما ترجع المؤشر لذلك الهدف.

أبسط خوارزميات البحث هي "البحث الخطي"، و الذي يمر في عناصر المجموعة بالترتيب و يتوقف عندما يجد الهدف. و في أسوأ الحالات فإن عليه أن يمر بكامل عناصر المجموعة. و لذلك فوقت التشغيل خطي.

المؤثر in للتسلسلات يستخدم بحثا خطيا، و كذلك طرائق المحارف مثل find و count.

إن كانت العناصر مرتبة في التسلسل، يمكنك استخدام البحث التنصيفي و الذي هو $O(\log n)$. البحث التنصيفي شبيه بالطريقة التي تبحث بها عن كلمة في القاموس (الحقيقي، و ليس هيكل البيانات). فبدلا من البدء من البداية و فحص كل عنصر بالترتيب، تبدأ من المنتصف و ترى إن كانت الكلمة التي تبحث عنها تأتي قبل أو بعد هذا الموقع. إن كانت تأتي قبل فستبحث في نصف النصف الاول. و إن كانت بعد فستبحث في نصف النصف الثاني. و أيا كان موقعها فقد اختصرت نصف البحث المتبقي في كل مرة.

إن كان في التسلسل 1000000 عنصر سيأخذ إيجاد الكلمة منك 20 خطوة، أو إثبات عدم وجودها. هذا يعني 50000 مرة أسرع من البحث الخطي.

تمرين ب-3 أكتب اقترانا اسمه bisection يأخذ قائمة مفروزة و قيمة مستهدفة و يرجع المؤشر لتلك القيمة في القائمة، إن كانت موجودة، أو يرجع None إن لم تكن.

أو يمكنك قراءة وثائق مديول bisection و استخدامه!

يمكن للبحث التنصيفي أن يكون أسرع بكثير من البحث الخطي، لكنه يتطلب كون التسلسل مرتبا، و هو ما يتطلب جهدا إضافيا.

هناك نوع اخر من هياكل البيانات تسمى تقطيعية hashtable، و هي أسرع من البحث التصنيفي – تستطيع القيام بالبحث في وقت ثابت – و لا تتطلب من العناصر أن تكون مرتبة. قواميس بايثون مطبقة باستخدام hashtables، و هو السبب الذي جعل كل عمليات القواميس و من ضمنها مؤثر in ثابتة الوقت.

ب-4 جداول التقطيع Hashtables

لشرح كيف تعمل جداول التقطيع و لماذا كان أداؤها بهذه الجودة، سأبدأ بتطبيق بسيط لخارطة ثم أحسنها تدريجيا إلى أن يصبح التطبيق تقطيعيا.

سأستخدم بايثون لشرح هذه التطبيقات، لكنك لن تكتب نصا كهذا في بايثون في الحياة الواقعية، ستستعمل القاموس فقط! لذلك و لنهاية هذه الفصل حاول أن تتخيل بأن القواميس غير موجودة و أنك تريد تطبيق هيكل للبيانات يوصل بين المفاتيح و القيم. العمليات التي ستطبقها هي:

Add(k, v): أضف عنصرا جديدا يوصل من المفتاح k إلى القيمة v. في قاموس بايثون d هذه العملية تكتب d[k] = v.

Get(target): تبحث عن القيمة التي تقابل المفتاح key و ترجعها. لقاموس بايثون d تكتب هذه العملية هكذا d[target] أو d.get(target).

سأفترض مؤقتا بأن كل مفتاح يظهر مرة واحدة. أبسط تطبيق لهذه الواجهة هو استخدام قائمة من التوبل، يكون كل توبل زوجين من المفتاح-القيمة.

```
class LinearMap(object):

   def __init__(self):
      self.items = []

   def add(self, k, v):
      self.items.append((k, v))

   def get(self, k):
      for key, val in self.items:
         if key == k:
            return val
      raise KeyError
```

تضيف add توبل مفتاح-قيمة لقائمة العناصر، و هو ما يأخذ وقتا ثابتا.

تستخدم get حلقة for للبحث في القائمة: فإن وجدت المفتاح المستهدف سترجع القيمة المقابلة له، و إلا فستصدر KeyError. إذن فـ get خطية.

هناك بديل هو إبقاء القائمة مفروزة حسب المفتاح. عندها تستطيع get استخدام البحث التنصيفي، و الذي هو $O(\log n)$. لكن إدراج عنصر جديد في وسط قائمة خطي، لذلك فقد لا يكون هذا هو الحل الامثل. هناك هياكل بيانات أخرى (أنظر:

http://en.wikipedia.org/wiki/Red-black_tree.) تطبق add و get في log time، لكن هذا أيضا ليس بجودة ثابتات الوقت، إذن لنتابع.

طريقة أخرى لتحسين LinearMap هي تفتيت قائمة أزواج المفتاح-القيمة إلى قوائم أصغر. هاك تطبيق اسمه BetterMap، و هو قائمة من 100 توصيل خطي LinearMap. سنرى حالا بأن ترتيب النمو لـ get لا زال خطيا، لكن BetterMap هي خطوة على طريق جداول التقطيع:

```
class BetterMap(object):

   def __init__(self, n=100):
      self.maps = []
      for i in range(n):
         self.maps.append(LinearMap())

   def find_map(self, k):
      index = hash(k) % len(self.maps)
      return self.maps[index]

def add(self, k, v):
      m = self.find_map(k)
      m.add(k, v)

def get(self, k):
      m = self.find_map(k)
      return m.get(k)
```

توجِد __init__ قائمة من LinearMap عددها n.

find_map تُستخدم من قبل add و get لتقرر في أي توصيل سيوضع العنصر الجديد، أو في أي توصيل ستبحث.

تستخدم find_map الاقتران الجاهز hash، و الذي يأخذ أي كائن بايثوني و يحوله لعدد صحيح. هذه العملية لها محدودية، و هي أنها تعمل مع المفاتيح التقطيعية. الانماط المتبدلة كالقواميس و القوائم غير تقطيعية.

الكائنات التقطيعية التي تعتبر متكافئة ترجع نفس قيمة التقطيع، لكن العكس ليس بالضرورة صحيح: كائنان مختلفان قد يرجعا نفس قيمة التقطيع.

يستخدم find_map مؤشر القسمة بدون باقي للف قيم التقطيع في مجال من صفر إلى len(self.maps)، إذن فالنتيجة هي مؤشر قانوني في القائمة. هذا يعني بالطبع أن الكثير من القيم التقطيعية ستلف في نفس المؤشر. لكن إن كان اقتران التقطيع يوزع الاشياء بالتساوي إلى حد ما (و هو ما صممت اقترانات التقطيع لأجله)، إذن بإمكاننا توقع n /100 عنصر في كل LinearMap.

و بما أن زمن التشغيل لـ LinearMap.get يتناسب مع عدد العناصر فيمكننا توقع أن BetterMap أسرع بمئة مرة من LinearMap. ترتيب النمو لا يزال خطيا، لكن المعامل القائد أصبح أصغر. لا بأس بهذه النتيجة، لكن جداول التقطيع Hashtable لازالت أفضل.

هنا (أخيرا) فكرة حاسمة تجعل جداول التقطيع أسرع: إن أمكنك إبقاء أطوال LinearMap ضمن إطار محدد، ستكون LinearMap.get ثابتة الوقت. كل ما عليك فعله هو متابعة عدد العناصرة و عندما يتجاوز عتبة تحددها، غير حجم جدول التقطيع بإضافة مزيد من LinearMap.

هنا تطبيق لجدول تقطيع:

```
class HashMap(object):
    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

كل HashMap تحتوي على BetterMap، تبدأ __init__ باثنتان من LinearMap ثم تهيئ num الذي سيتابع عدد العناصر.

يوفد get إلى BetterMap . الشغل الحقيقي سيقوم به add الذي يتحقق من عدد العناصر و من حجم BetterMap: إن تساوا، فإن معدل عدد العناصر في كل LinearMap يكون 1، إذن سينادي على resize.

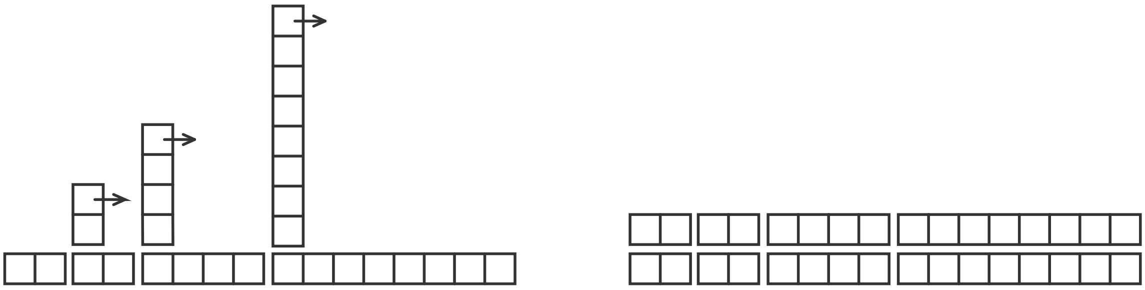
resize تنشئ BetterMap جديدة، بضعفي حجم السابقة، ثم "تعيد تقطيع" العناصر من الخارطة السابقة إلى الجديدة.

من المهم إعادة التقطيع لأن تغيير عدد الـ LinearMap يغير مقام مؤثر القسمة بدون باقي في find_map. هذا يعني بأن بعض الكائنات التي كانت ملفوفة في نفس الـ LinearMap ستقسم (و هو ما أردناه، ألا توافق؟).

إعادة التقطيع خطية، و عليه فإن resize خطية، و هو ما قد لا يبدو جيدا بما أنني وعدت بأن add ستكون ثابتة الوقت. لكن تذكر بأنه ليس علينا إعادة ضبط الحجم في كل مرة، إذن ف add ثابتة الوقت في العموم و أحيانا فقط خطية. كمية العمل الكلية لتشغيل add عدد *n* من المرات يتناسب مع *n*، إذن متوسط الوقت لكل add يصبح ثابت الوقت!

لكي ترى كيف تعمل هذه، فكر في البدء بجدول تقطيع فارغ ثم إضافة تسلسل من العناصر. سنبدأ بإثنتين LinearMap، إذن فعمليتي add الاولتين سريعتين (لإنه لا توجد حاجة لإعادة ضبط الحجم). لنقل أنهما تأخذا وحدة واحدة من الشغل لكل منهما. add التالية تتطلب إعادة ضبط الحجم، إذن فعلينا إعادة تقطيع العنصرين الأولين (لندعو هذا وحدتين إضافيتين من الشغل) ثم أضف عنصرا ثالثا (وحدة شغل ثالثة). إضافة العنصر التالي يكلفنا وحدة واحدة، إذن فالمجمل حتى الان 6 مديولات من الشغل لأربعة عناصر.

الـ add التالية تكلف 5 وحدات، لكن الثلاثة التي تلي تكلف وحدة واحدة لكل منها، إذن فالإجمالي هو 14 وحدة لأول 8 عمليات add.



الشكل ب-1: تكلفة جداول التقطيع لـ add

ستكلف add التالية 9 وحدات، لكن بعدها يمكننا إضافة 7 add قبل عملية ضبط الحجم التالية، إذن المجموع هو 30 وحدة لأول 16 add.

بعد 32 عملية add ستكون التكلفة الاجمالية 64 وحدة، و أتمنى الان أن النمط أصبح واضحا لديك. و بعد عدد n من add حيث n هي للقوة 2، سيصبح مجموع الكلفة هو $2n$-2 وحدة، إذن فمتوسط الشغل لـ add واحدة هو أقل من وحدتين. عندما تكون n للقوة 2 تكون أفضل الحالات، لقيم أخرى لـ n سيكون متوسط الشغل أعلى قليلا، لكن هذا ليس بذي أهمية. المهم هو أنها $O(1)$.

الشكل ب-1 يوضح كيف يعمل هذا الامر بالرسوم. كل مربع يمثل وحدة شغل. الاعمدة تبين الشغل الكلي لكل add بالترتيب من اليسار إلى اليمين: عمليتي add الأولتين كلفتا وحدة واحدة، الثالثة كلفت ثلاث وحدات، إلخ.

عمليات إعادة التقطيع الإضافية تظهر كتسلسل من الابراج المتطاولة و كذلك المتباعدة فيما بينها باضطراد. الآن إن هدمت الأبراج مسددا بها تكلفة إعادة التقطيع الكلية، سترى بالرسم أن التكلفة الكلية بعد عدد n من الـ add هو $2n$-2,

لهذه الخوارزمية خاصية مهمة وهي أننا عندما نضبط حجم جدول التقطيع فإنه ينمو كمتوالية هندسية، أي أننا نضرب الحجم بثابت. فإن زدت الحجم حسابيا –بإضافة عدد ثابت في كل مرة – سيكون متوسط الوقت لكل add خطيا.

يمكنك تحميل تطبيقي لـ HashMap من `http://thinkpython.org/code/Map.py`، لكن تذكر بأنه لا يوجد ما يستدعي استخدامه، إن أردت التوصيل فاستخدم قاموس بايثون.

الملحق ج

لمبي Lumpy

استخدمت، على طول الكتاب، رسومات لتمثيل حالة برنامج شغال.

في القسم 2.2 استخدمنا رسم الحالة لبيان أسماء و قيم المتغيرات. و في القسم 3.10 قدمت لك الرسم التستيفي، الذي يعرض إطارا واحدا لكل نداء لاقتران. كل إطار يعرض البرمتر و المتغير المحلي للإقتران أو الطريقة. الرسوم التستيفية للإقترانات الإجترارية ظهرت في القسمين 5.9 و 6.5.

القسم 10.2 بيّن كيف تظهر القائمة في رسم الحالة، و في القسم 12.6 عرفنا طريقتين لتمثيل التوبل.

القسم 15.2 قدم لنا رسم الكائن، و الذي يظهر حالة خصال كائن، و خصالها و خصال خصالها و هكذا، القسم 15.3 كا فيه رسوم الكائن للمستطيلات و نقاطها المضمنة. القسم 16.1 بين الحالة لكائن Time. و في القسم 18.2 كان الرسم الذي يحتوي على كائن فئة و تجلية، كل منهامع خصالها.

أخيرا، القسم 18.8 قدم رسم الفئة، و الذي أظهر الفئات التي بني عليها البرنامج و العلاقات بينها.

كل هذه الرسومات مبينية على "لغة النمذجة الموحدة" UML: Unified Modeling Language و هي لغة رسومية مقننة يستخدمها مهندسو الحاسوب للتواصل فيما بينهم حول تصميم البرامج، خصوصا البرامج كائنية المنحى.

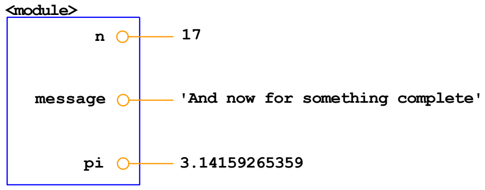
UML لغة غنية بمختلف أشكال الرسومات، التي تمثل العديد من العلاقات بين الكائنات و الفئات. ما قدمته في هذا الكتاب ما هو إلا جزء يسير من هذه اللغة لكنه كان الجزء الأكثر استعمالا في الحياة العملية.

هدف هذا الملحق هو استعراض الرسومات في الاقسام السابقة و تقديم لمبي Lumpy . لمبي (و هي اختصار لـ UML Python مع تبديل مواقع بعض الحروف)، هي جزء من سوامبي، الذي سبق و نصّبته إن كنت قد عملت على دراسة الحالة في الفصل الرابع أو الفصل التاسع عشر، أو إن عملت على التمرين 15.4.

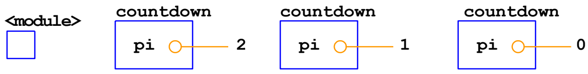
يستخدم لمبي مديول بايثون `inspect` لفحص حالة برنامج شغال و ليولد رسومات الكائن (و منها الرسم التستيفي) و كذلك رسومات الفئات.

ج-1 رسم الحالة

هنا مثال يستخدم لمبي لتوليد رسم حالة



الشكل ج-1 رسم حالة رسم مع لمبي



الشكل ج-2 رسم تستيفي

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

message = 'And now for something completely different'
n = 17
pi = 3.1415926535897932

lumpy.object_diagram()
```

يستورد السطر الأول فئة لمبي من swampy.Lumpy. إن لم يكن سوامبي منصب كحزمة لديك فتأكد من أن ملفات سوامبي موجود مسار بحث بايثون و استخدم عبارة الاستيراد التالية:

```
from Lumpy import Lumpy
```

ثم تولد السطور التالية كائن lumpy و توجد نقطة "مرجعية"، مما يعني بأن لمبي يسجل الكائنات التي عرفت لحد الان.

في التالي نعرِّف متغيرات جديدة و نستدعي object_diagram الذي سيرسم الكائنات التي تم تعريفها منذ النقطة المرجعية، هنا كانت message, n, pi.

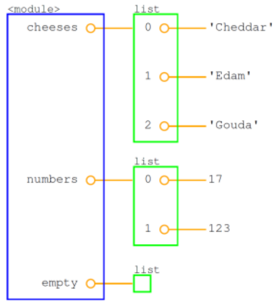
الشكل ج-1 يظهر النتيجة. يختلف أسلوب الرسم عما أريتك إياه سابقا، مثلا هنا كل مرجع ممثل بدائرة بالقرب من اسم المتغير و خط يصل إلى القيمة. و الحارف الكبيرة مبتورة. لكن تظل المعلومات المستقاة من الرسم نفسها.

الاسماء العمومية موضوعة في إطارٍ ملصَقه <module>، و هو يشير إلى أن هذه متغيرات على مستوى المديول، و تدعى أيضا عمومية.

يمكنك تحميل هذا المثال من http://thinkpython.org/code/lumpy_demo1.py. حاول إضافة تعيينات أخرى و لاحظ كيف سيبدو الرسم.

ج-2 الرسم التستيفي

التالي هو مثال يستخدم لمبي لتوليد رسم تستيفي، و يمكنك أيضا تحميله من http://thinkpython.org/code/lumpy_demo2.py.



الشكل ج-3 رسم الكائن

```
from swampy.Lumpy import Lumpy

def countdown(n):
    if n <= 0:
        print 'Blastoff!'
        lumpy.object_diagram()
    else:
        print n
        countdown(n-1)

lumpy = Lumpy()
lumpy.make_reference()
countdown(3)
```

الشكل ج-2 يبين النتيجة. يمثل كل إطار بصندوق يكون اسم الاقتران خارجه و المتغير بداخله. و بما أن هذا الاقتران إجتراري، يكون هناك إطار لكل مرحلة من الإجترارات.

تذكر بأن الرسم التستيفي يظهر حالة البرنامج عند نقطة محددة في تنفيذه. و لتحصل على الرسم الذي تريده، أحيانا عليك التفكير في المكان الذي يجب أن تستدعي object_diagram منه.

و هنا استدعيت object_diagram مباشرة بعد تنفيذ الحالة القاعدة للإجترار، و في هذه الحالة سيريني الرسم التستيفي كل مرحلة من مراحل الاجترار. يمكنك استدعاء object_diagram أكثر من مرة لتحصل على لقطات من مراحل تنفيذ البرنامج.

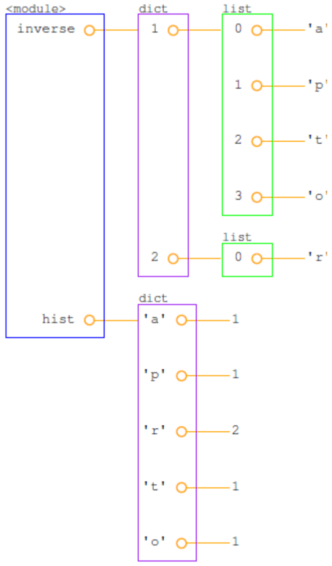
ج-3 رسوم الكائن

هذا المثال يولد رسم كائن يعرض القوائم في القسم 10.1. يمكنك تحميله من http://thinkpython.org/code/lumpy_demo3.py.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

cheeses = ['Cheddar', 'Edam', 'Gouda']
```



الشكل ج-4: رسم الكائن.

```
numbers = [17, 123]
empty = []
lumpy.object_diagram()
```

الشكل ج-3 يبين النتيجة. القوائم ممثل بصناديق تظهر توصيل المؤشرات إلى العناصر. هذا التمثيل خادع قليلا، و ذلك لأن المؤشرات ليست جزءا من القائمة في الحقيقة، لكنه في رأيي يجعل الرسم أسهل للقراءة. القائمة الفارغة هنا تمثل بصندوق فارغ.

و هنا مثال يعرض القواميس من القسم 11.4 . يمكنك تحميله من .http://thinkpython.org/code/lumpy_demo4.py

```
from swampy.Lumpy import Lumpy

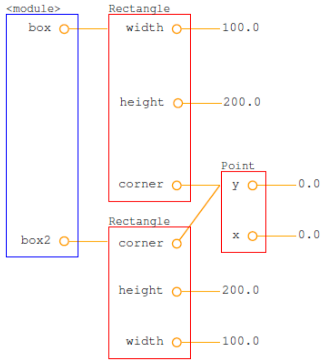
lumpy = Lumpy()
lumpy.make_reference()

hist = histogram('parrot')
inverse = invert_dict(hist)

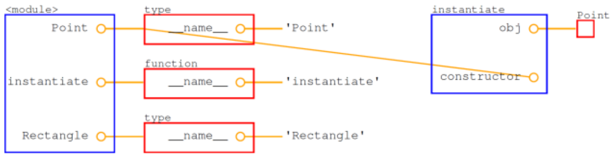
lumpy.object_diagram()
```

الشكل ج-4 يبين النتيجة. يصل القاموس hist بين الحروف (محارف من حرف واحد) و بين الأعداد الصحيحة، و inverse يصل من الاعداد الصحيحة إلى قوائم المحارف.

و هذا المثال يولد رسم كائن لـ Point و Rectangle. كما في القسم 15.6. يمكنك تحميله من .http://thin;python.org/code/lumpy_demo5.py



الشكل ج-5: رسم الكائن



الشكل ج-6: رسم الكائن

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

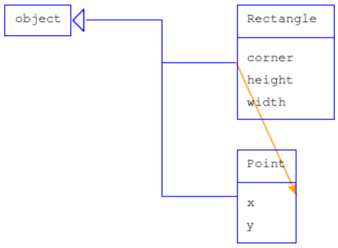
box2 = copy.copy(box)

lumpy.object_diagram()
```

الشكل ج-5 يظهر النتيجة. توجد `copy.copy` نسخة ضحلة، فيكون لكل من `box` و `box1`طوله و عرضه الخاص `width` و `height`. لكنهما يتشاركان نفس كائن Point المضمن. لا بأس بهذا النوع من التشارك عندما تكون الكائنات غير متبدلة، لكن للأنماط المتبدلة فهو مصدر كبير للأخطاء.

ج-4 كائنات الاقترانات و الفئات

عندما أستخدم لمبي لعمل رسوم الكائنات، فأنا في العادة أعرِّف الاقترانات و الفئات قبل عمل النقطة المرجعية. و بهذه الطريقة لا تظهر كائنات الاقتران و الفئة في الرسم.



الشكل ج-7 رسم الفئة

لكن إن كنت تمرر الاقتران و الفئات كبرمترات، فقد تريد لها أن تظهر في الرسم. هذا المثال يبين كيف ستبدو، يمكنك تحميله من http://thinkpython.org/code/lumpy_demo6.py.

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

class Point(object):
    """Represents a point in 2-D space."""

class Rectangle(object):
    """Represents a rectangle."""

def instantiate(constructor):
    """Instantiates a new object."""
    obj = constructor()
    lumpy.object_diagram()
    return obj

point = instantiate(Point)
```

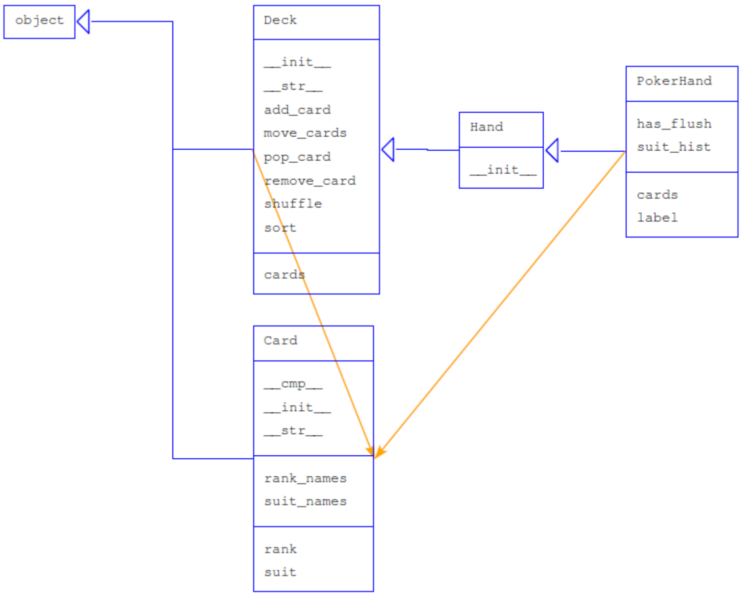
يبين الشكل ج-6 النتيجة. بما أننا نستدعي object_diagram داخل اقتران، سنحصل على رسم تستيفي بإطار للمتغيرات على مستوى المديول و لاستدعاء instantiate.

على مستوى المديول يكون مرجع Point و Rectangle كائن فئة (و الذي نمطه type)، أما instantiate فمرجعه كائن اقتران.

قد يوضح الرسم نقطتان ملتبستان شائعتان: (1) الفرق بين كائن الفئة Point و التجلية obj التي هي لـ Point. و (2) الفرق بين كائن الاقتران الذي أنشئ عندما عرف instantiate و بين الاطار الذي أنشئ به عندما نودي.

ج-5 رسم الفئة

رغم أنني أفرّق بين رسم الحالة و الرسم التستيفي و رسم الكائن، إلا أنها في الغالب نفس الشيء: فهي تظهر حالة البرنامج الشغال في لحظة ما من الزمن.



الشكل ج-8 رسم الفئة

رسوم الفئة مختلفة. فهي تظهر الفئات التي يقوم البرنامج عليها و العلاقات بينها. فهي لا علاقة لها بالوقت من حيث أنها تصف البرنامج ككل و ليس في لحظة زمنية معينة. مثلا إن كانت تجلية من الفئة A تحتوي على مرجع لتجلية من الفئة B سنقول أنها العلاقة بين هاتين الفئتين هي علاقة HAS-A.

هذا مثال يبين علاقة HAS-A. يمكنك تحميله من

.http://thinkpython.org/code/lumpy_demo7.py

```
from swampy.Lumpy import Lumpy
lumpy = Lumpy()
lumpy.make_reference()

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

lumpy.class_diagram()
```

الشكل ج-7 يظهر النتيجة. كل فئة يمثلها صندوق يحتوي اسم الفئة و أي الطرائق توفر و أي متغيرات فئة و أي متغيرات تجلياتها. في هذا المثال كان لـ Rectangle و Point مغيرات تجلية، لكن لم يكن لها متغيرات فئة و لا طرائق.

السهم من Rectangle إلى Point يبين بأن في المستطيلات نقطة مضمنة. و يبين أيضا بأن Rectangle و Point ترثان من object، و الذي بدوره يمثل في الرسم بالسهم ذو الرأس المثلث.

إليك الآن مثال أكثر تعقيدا باستخدام حلي للتمرين 18.6 . تستطيع تحميله من

http://thinkpython.org/code/lumpy_demo8.py

ستحتاج أيضا إلى http://thinkpython.py/code/PokerHand.py.

```
from swampy.Lumpy import Lumpy
```

```
from PokerHand import *
lumpy = Lumpy()

lumpy.make_reference()
deck = Deck()

hand = PokerHand()
deck.move_cards(hand, 7)
lumpy.class_diagram()
```

الشكل ج-8 يبين النتيجة. PokerHand ترث من Hand ، و التي ترث من Deck. و لكل من Deck و PokerHand بطاقات لعب.

لا يُظهر هذا الرسم بأن لـ Hand بطاقات أيضا، لأنه لا توجد تجليات في البرنامج لـ Hand . يبرز هذا المثال محدودية للمبي، فهو يعلم فقط عن الخصال و علاقات HAS-A بين الكائنات التي أوجدت لها تجليات.